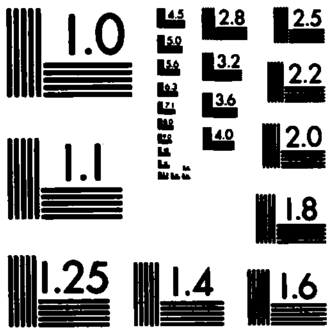

1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

①

AD-A134 032

CONTRACT F30602-80-C-0291

IR-677-2 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR Ada INTEGRATED ENVIRONMENT: Ada COMPILER PHASES B5-AIE (1) COMP (1)

5 NOVEMBER 1982

PREPARED FOR: ROME AIR DEVELOPMENT CENTER
CONTRACTING DIVISION/PKRD
GRIFFISS AFB, N.Y. 13441

PREPARED BY: INTERMETRICS, INC.
733 CONCORD AVE.
CAMBRIDGE, MA 02138

83 09 19 053

B5-AIE(1).COMP(1)

This document was produced under contract F30602-80-C-0291/SA P0009 for the Rome Air Development Center. Mr. Donald Mark is the Program Engineer for the Air Force. Mr. Mike Ryer is the Project Manager for Intermetrics.

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
PER LETTER	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

TABLE OF CONTENTS

Table of Contents (con't)

Table of Contents (con't)

Table of Contents (con't)

Table of Contents (con't)

Table of Contents (con't)

1.0 SCOPE

1.1 Identification

This document specifies the requirements for the performance and verification of the Ada compilers (COMP) for the IBM (VM/370) and Perkin-Elmer (PE) 8/32 (OS/32) systems. Each compiler provides the user with the ability to translate an Ada compilation and obtain a program listing and linkable machine code for the respective target machine; listing, optimization, and debugging control are selectable by the user. Because of the compiler structure and the similarity of the target machines, the two compilers are nearly identical. As a result, this document presents the design as though there were a single Ada compiler; where target-machine dependencies make the compilers different, this is pointed out in the discussion.

The CPCI's that comprise the compiler subsystem are listed below along with their component CPC's. An asterisk indicates two CPCI's (one for the IBM 370/VM and one for the PE 8/32).

CPCIname	CPCname
(ID)	(ID)

FRONT END(FE)	DRIVER(A)
	LEXSYN(B)
	SEM(C)
	UTILITIES(D)

*MIDDLE PART(MID)	GENINST(A)
	STATINFO(B)
	STORAGE(C)
	EXPAND(D)
	UTILITIES(E)

*BACK END (BE)	FLOW(A)
	VCODE(B)
	TNBIND(C)
	CODEGEN(D)
	FINAL(E)
	UTILITIES(F)

DIANA (DIANA)	
LOW-LEVEL INTERMEDIATE LANGUAGE (BILL)	

1.2 Functional Summary

The Ada compiler is composed of several phases, partitioned into a Front End, a Middle Part, and a Back End. The Front End is organized into two processing phases that, together, perform lexical, syntactic, and semantic analysis and generate a DIANA representation for each compilation unit. The compiler DRIVER (COMP.FE.A) that selects processing phases is also part of the Front End. The Middle Part, organized into four processing phases, selects the run-time model and produces a low-level tree representation incorporating machine-dependent decisions. The Back End, organized into five processing phases, performs optimization and code generation and yields a linkable object program.

Language dependencies in the compiler are concentrated in the Front End (static semantics) and the Middle Part (run-time semantics). There are relatively few language dependencies in the Back End. Target machine dependencies occur in the Middle Part and the Back End. Though the Front End may have to call machine-dependent procedures during semantic analysis, the interface is narrow and clearly defined.

Figure 1-1 shows the phase breakdown. The compiler phases are strictly sequential and may be overlaid for host systems lacking virtual memory facilities.

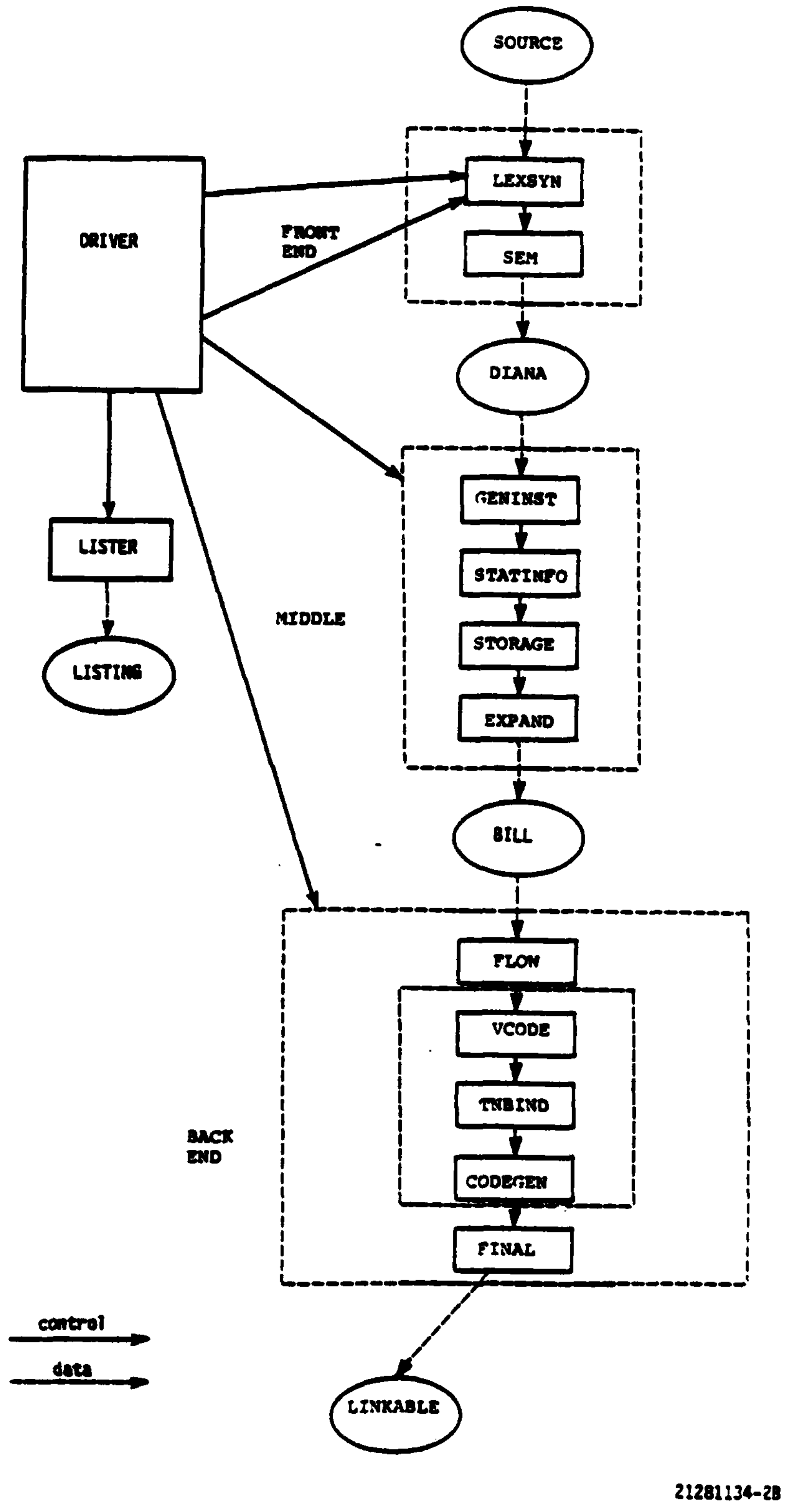


FIGURE 1-1: Compiler Phase Breakdown

2.0 APPLICABLE DOCUMENTS

2.1 Program Definition Documents

Reference Manual for the Ada Programming Language, Draft proposed ANSI standard document, July 1982.

Requirements for Ada Programming Support Environments, "STONEMAN", February 1980, Department of Defense.

Revised Statement of Work, (15 March 1980).

2.2 Inter Subsystem Specifications

System Specification for Ada Integrated Environment, Type A

Computer Program Development Specifications for Ada Integrated Environment (Type B5):

KAPSE/Database, AIE(1).KAPSE(1)

MAPSE Command Processor, AIE(1).MCP(1)

MAPSE Generation and Support, AIE(1).MGS(1)

Program Integration Facilities, AIE(1).PIF(1)

MAPSE Debugging Facilities, AIE(1).DBUG(1)

MAPSE Text Editor, AIE(1).TXED(1)

Virtual Memory Methodology, AIE(1).VMM(2)

Technical Report (INTERIM), IR-684

2.3 Military Specifications and Standards

Data item description DIE-30139, USAF, 24 July 1976.

2.4 Miscellaneous Documents

Diana Reference Manual, G. Goos and Wm. Wulf, Institut fuer Informatik II, Universitaet Karlsruhe and Computer Science Dept., Carnegie Mellon University, March 1981.

SIGPLAN NOTICES, Volume 17, November 6, June 1982. A Pratical Method for Syntactic Error Recovery Diagnosis and Recovery. Micheal Burke and Gerald Fisher, Courant Institute, New York University, 251 Mercer Street, New York, N.Y. 10012. Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction.

NYU LALR Parser Generator, Philippe Charles and Gerald Fisher, Courant Institute, New York University, unpublished paper, 1981.

J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, B.A. Wichmann, Rationale for the Design of the Ada Programming Language; ACM SIGPLAN Notices Vol. 14, No. 6, June 1979, Part B.

G. Persch, G. Winterstein, M. Dausmann, S. Drossopoulou, "Overloading in Preliminary Ada," in ACM SIGPLAN Notices Vol 15, No. 11., November 1980, pp. 47-56.

J. Welsh, "Economic Range Checks in Pascal", Software Practice and Experience, Vol. 8, No. 1, January 1978, pp. 85-98.

R. Firth, Notes on Range Checking in Ada, Workshop on Intermediate Languages, Murnau, W. Germany, February 1981.

W.A. Wulf "PQCC: A Machine-Relative Compiler Technology" Carnegie-Mellon University, Department of Computer Science, 25 September 1980.

J.B. Goodenough, "The Ada Compiler Validation Capability", in SIGPLAN Notices, Vol. 15, No. 11, November 1980, pp. 1-8.

R.G. Scarborough and H.G. Kolsky "Improved Optimization of FORTRAN Object Programs", IBM Journal of Research and Development, Vol 24, No 6, Nov. 1980, pp. 660-676.

R. Cattell "Formalization and Automatic Generation of Code Generators" Ph.D. Thesis, Carnegie-Mellon University, 1978.

P.F. Stockhausen "Adapting Optimal Code Generation for Arithmetic Expressions to the Instruction Sets Available on Present-Day Computers", Comm. ACM, Vol. 15, No. 6, June 1973, pp. 353-354.

R. Sethi and J.D. Ullman "The Generation of Optimal Code for Arithmetic Expressions", Journal ACM, Vol. 17, No. 4, October, 1970, pp. 715-728.

B.M. Brosgol, "An Implementation of ECL Data Types", in SIGPLAN Notices, Vol. 6, No. 12, December 1971, pp. 87-95.

J. Cocke and J.T. Schwartz, Programming Languages and Their Compilers, Courant Institute of Mathematical Sciences, New York Univ., April 1970.

B. Leverett, Register Allocation in Optimizing Compilers. Ph. D. Thesis, Carnegie-Mellon Univ., February 1981.

3.0 REQUIREMENTS

3.1 Introduction

This section provides the set of requirements for the AIE compiler. This includes the performance and interface specifications to which the compiler must comply.

3.1.1 General Description

The compiler is a tool in the MAPSE toolset which converts Ada source text into machine code to execute on a target machine. The compiler operates in three major pieces, invoked in order from DRIVER. The pieces are the Front End, the Middle Part, and the Back End.

The Front End of the compiler checks the source for compliance with the rules of the Ada language definition, including syntactic and semantic rules. The Front End inputs Ada source and outputs DIANA as an intermediate language.

The Middle Part determines run-time storage requirements, gathers statistics from the DIANA program, and converts the DIANA program into a lower-level representation called BILL.

The Back End optimizes the BILL code, and converts it into machine code.

Other MAPSE tools are described in the documents listed in Section 2 2.

3.1.2 Peripheral Equipment Identification

Not applicable.

3.1.3 Interface Identification

Figure 3-1 shows the relationships of the compiler to other parts of the AIE. Program interfaces are described in detail in Section 3.2.4.

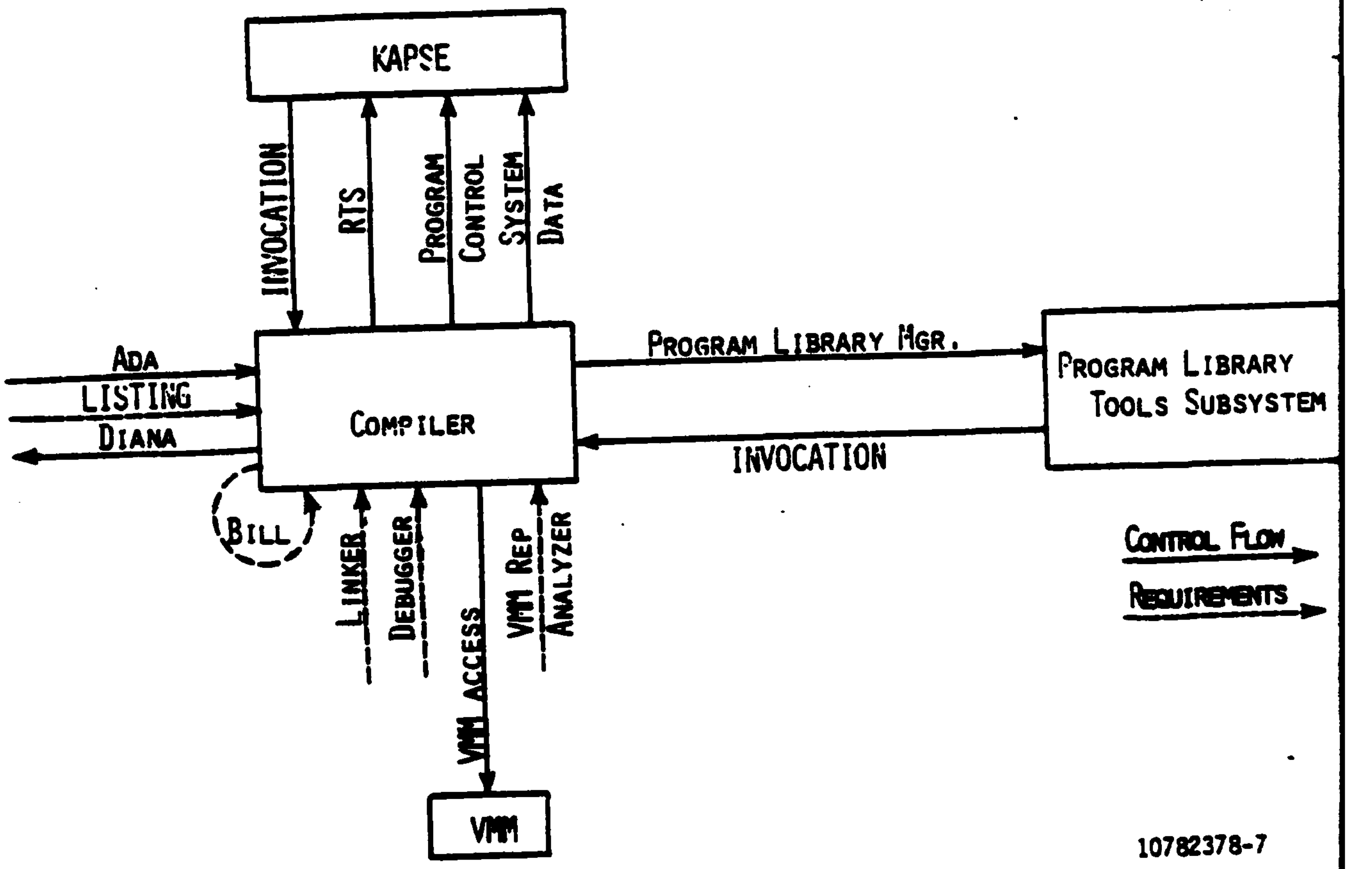


FIGURE 3-1: Compiler Interfaces

3.2 Functional Description

3.2.1 Equipment Description

Not applicable.

3.2.2 Computer Input/Output Utilization

Not applicable.

3.2.3 Computer Interface Block Diagram

Not applicable.

3.2.4 Program Interfaces

3.2.4.1 KAPSE Interface

The KAPSE interface provides program invocation control system data, and Ada run-time support (KAPSE.MULTPROG and KAPSE.RTS).

All Ada programs, of which the compiler is one, use the run-time system of the KAPSE. In addition, the compiler uses program control to access its parameters, to handle control over compiler phases, and to invoke the Lister (LISTER). The compiler also uses the KAPSE to provide statistics on compilation.

The compiler, being written in Ada, will, while executing call upon facilities provided for all Ada programs in the run-time system. These facilities include I/O, heap management, and exception handling, but do not include tasking.

The compiler uses the KAPSE to bring in various compiler phases either by invoking a separate program underneath DRIVER, or overlaying a successor phase on top of a predecessor. The compiler may be invoked by the program library tools (PIF.PLTOOLS), as well as invoke program library tools (e.g., LISTER). The KAPSE primitive assumed is: "call separate program with parameters".

To provide statistics and information to be incorporated in the listing, the compiler uses the KAPSE. This information is:

(1) get current date and time
(2) get user id
(3) query cpu clock

3.2.4.2 Program Library Interface

The compiler uses the Program Library (PIF.PLTOOLS) to access and store results of compilations, and to provide services needed for recompilation and separate compilation. The functions provided allow one to:

(1) Access a program library (including creating a new one if needed)
(2) Check existence of a program library
(3) Get library modes
(4) Set library modes
(5) Add objects to library
(6) Delete objects from library
(7) Find objects in library given Ada name and distinguishing attributes
(8) Get object attributes
(9) Set object attributes

3.2.4.3 Compiler Data Interface

The various compiler sections and phases must conform to agreed upon formats in order to pass information forward from phase to phase, and enable separate compilation. DIANA is an internal representation visible to other tools in the AIE, while BILL is strictly an internal representation.

3.2.4.3.1 DIANA Format

The compiler shall conform to DIANA as documented in AIE(1).COMP(1).DIANA(1).

3.2.4.3.2 BILL Format

The compiler shall conform to BILL as documented in AIE(1).COMP(1).BILL(1).

3.2.4.4 Virtual Memory Methodology (VMM) Interface

Compilation and separate compilation are done using KAPSE objects as extended core memory in a software paging system called VMM (VMM.VMM). There are two parts of VMM used by the compiler: the Rep Analyzer (VMM.VMM.A), and the VMM access routines.

The Rep Analyzer is given the specifications of data structures to be paged in a particular compiler phase, and generates Ada source for procedures which create, modify, and access instances of those data structures.

The procedures generated by the Rep Analyzer are combined with routines provided in the VMM package to enable the compiler to access and manipulate paged data structures, both those defined by the user, and those predefined abstractions provided by VMM.

3.2.4.4.1 Rep Analyzer

The compiler shall provide appropriate input for the Rep Analyzer. The DIANA definition [AIE(1).COMP(1).DIANA(1)] is run through a tool that converts it into legal Ada format.

3.2.4.4.2 VMM Access Routines

The compiler will use VMM support specifications to do the following:

(1) open and close a domain
(2) access and close a subdomain (including new creation of a subdomain)
(3) destroy a subdomain
(4) create a node
(5) reference a node (get and set of value)
(6) change a node kind
(7) get and set root node of a subdomain
(8) use predefined abstractions (lists, sets, strings)

3.2.4.5 Listing Interface

The compiler will conform to the interface specified for LISTER as specified in [(AIE(1).PIF(1)].

3.2.4.6 Parameter Interface

The compiler can be invok with a variety of parameters to control its processing, and may also be supplied with parameters to pass on to the program library manager (PIF.PLTOOLS).

The user's request to compile Ada source is represented as:

```
COMPILE [SOURCE=> text_file] [LIBRARY=> prog_lib] [option=>value...]
```

The text file containing the source is identified by the optional SOURCE=> parameter. If the parameter is omitted, the source is read from the standard input file.

The program library to be used is specified with the LIBRARY=> parameter. If omitted, the COMPILE request is interpreted as a request for a syntax check with no semantic processing and no other permanent output.

Options may be specified with the COMPILE request. These options and their values are identified below.

LIST=> {ON, OFF, SOURCE, NOSOURCE, ATTRS, NOATTRS, XREF, NOXREF, ASSEMBLY, NOASSEMBLY}

The LIST parameter may be given a list of keywords whose default value the user wishes to change. These are:

ON	enable a listing
OFF (default)	disable listing
SOURCE (default)	list text
NOSOURCE	do not list text
ATTRS	list symbol table attributes of identifiers
NOATTRS (default)	do not list attributes
XREF	provide cross-reference of all identifiers
NOXREF (default)	do not provide cross reference of all identifiers
ASSEMBLY	list generated code
NOASSEMBLY (default)	no generated code
LISTERRS=>n	print errors above severity n in listing. Default is 0.
TTYERRS=>n	print errors above severity n on the terminal. Default is 0.
NOSEM=>n	if more syntax errors than n occur, suppress all phases of the compiler after the parser. Default is 50.
NOCODE=>n	if more semantic errors than n occur, suppress all phases of the compiler after semantics. Default is 50.
DEBUG=>{ALTER, NOALTER, BREAK, NOBREAK}	ALTER allows DBUG to alter and inspect information by preventing the optimizer from detecting common subexpressions across statement boundaries, and moving loads and stores of variables across statement boundaries. This enables DBUG to access variables which it might otherwise not be able to access. Default is NOALTER. BREAK inserts DBUG hooks after each statement and at the beginning and end of each procedure, so that a breakpoint may be effected easily. NOBREAK is the default.

OPTIMIZE=>{SPACE, TIME, NONE}	Only one optimize option may be selected. This is the same as the OPTIMIZE PRAGMA. Default is TIME.
STATISTICS=>{ON,OFF}	What statistics will be available is not yet decided. Finer control over which statistics are to be printed will also be available later.
COMMENT=> {ON,OFF}	COMMENTS=> ON preserves comments in the DIANA, enabling more complete source reconstruction. Default is OFF.
REORDER=> {ON,OFF}	REORDER=>ON allows the compiler to reorder compilation of units. Default is ON.
SPACE=> n	Allows the compiler n kilobytes of space in which to fit. Default is 512. The minimum and maximum values are not currently known. It is used to fit the compiler in as small a space as possible, or use large amounts of memory to improve the speed of compilation of large programs.
LOOKAHEAD=>n	n is the number of tokens to look ahead in parsing for a valid syntactic error recovery. Default is 5.
TRACE=> {ON,OFF}	Turn on tracing within the compiler. Used by compiler developers and maintenance only.

3.2.4.7 Debugger Interface

In order to allow the debugger [DBUG.DBUG] to function successfully, some information will have to be left in the DIANA tree and some additional tables may have to be generated and stored in the program library. For a full specification of the required interface, see AIE(1).DBUG(1).

3.2.4.8 Linker Interface

The compiler will conform to the linker interface, as specified in AIE(1).PIF(1).

3.2.4.9 Invocation Interface

The compiler is invoked by the KAPSE and may invoke LISTER (PIF.PLTOOLS.A). The compiler may also be recursively invoked by the (PIF.LINK) when a program is not up-to-date in the program library and the user wishes to execute it. Furthermore, the compiler may be invoked by the program library interface to assist in updating the library. In all cases, the compiler shall conform to the invocation specifications provided by the KAPSE.

3.2.4.10 Ada Interface

The compiler shall conform to the Ada language standard in two ways. It shall accept as valid input only Ada programs, and, being written in Ada, it shall itself be a valid Ada program.

The compiler will use a variety of the language features, including predefined packages, but excluding tasking.

3.2.5 Function Description

Figure 3-2 shows the flow of control in the compiler. Figure 3-3 shows the flow of data.

3.2.5.1 Front End

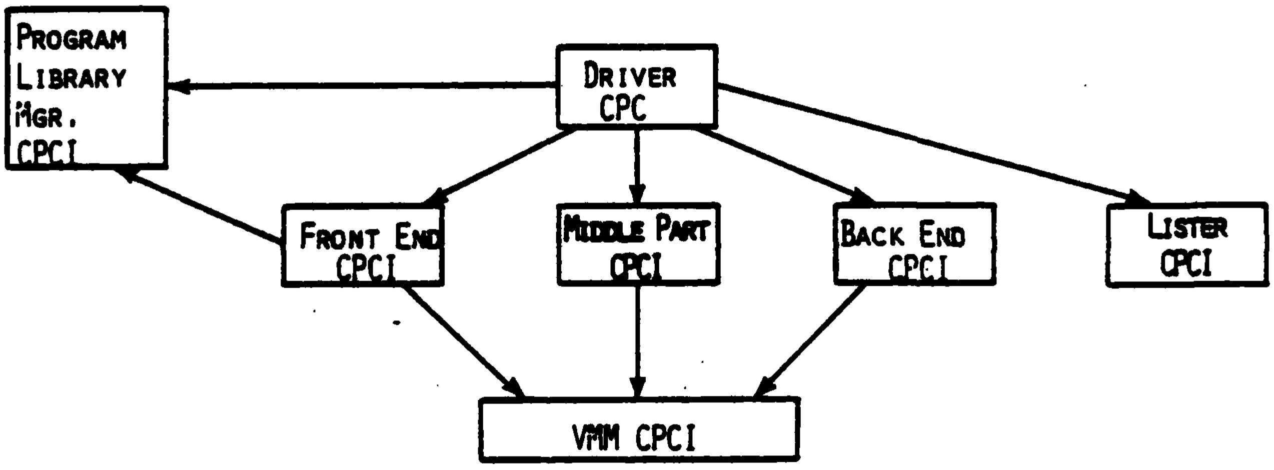
The compiler Front End consists of four components: DRIVER, LEXSYN, SEM, and UTILITIES.

Figure 3-4 shows the flow of control in the Front End. Figure 3-5 shows the flow of data in the Front End.

The DRIVER is responsible for coordinating the phases of the compiler and providing the appropriate working environment for them. The DRIVER provides VMM domains to a phase, and releases subdomains to the library when a phase successfully terminates.

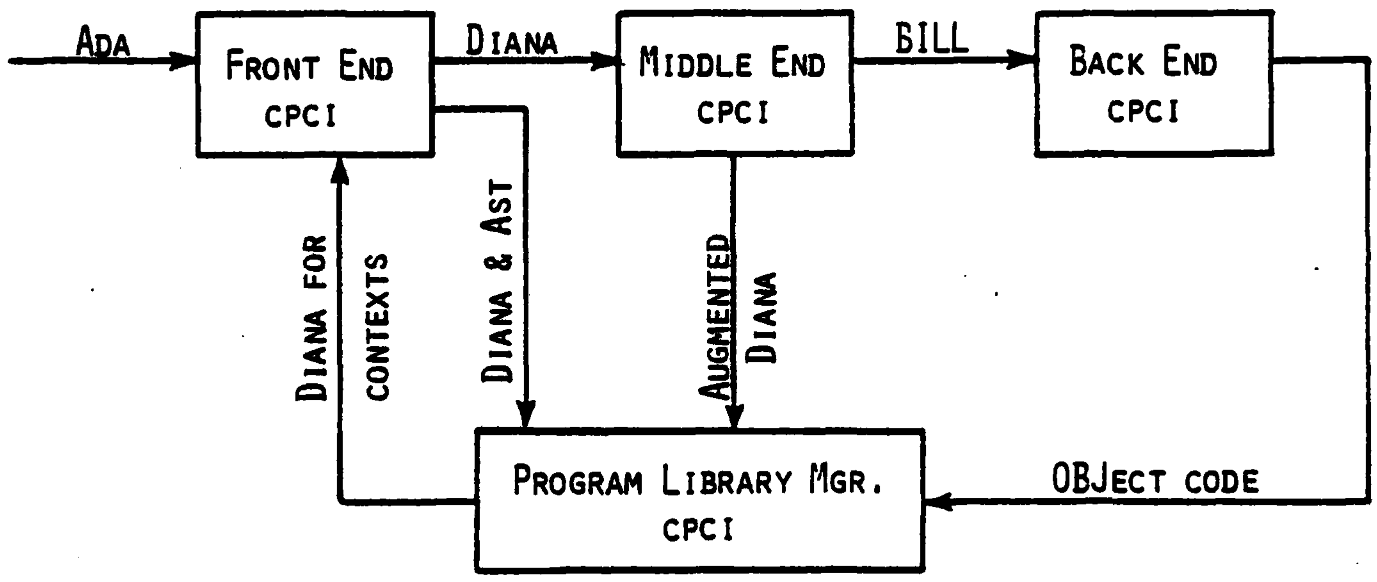
The LEXSYN phase performs lexical and syntactic analysis. It reads the source text for the compilation and produces an abstract syntax tree, using a bottom-up parse algorithm driven by LR tables. The LEXSYN phase also produces a name table as a preliminary for SEM to create a complete symbol table. In addition, LEXSYN performs pre-semantic checking on the tree being produced. The checks performed are semantic tests which depend only upon the content of the abstract syntax tree, and not upon the use of any symbol table information. LEXSYN produces a tree of compilation_unit nodes. After LEXSYN, each phase acts in turn on a single compilation unit.

The SEM phase performs semantic analysis for a compilation unit and transforms the abstract syntax tree into a DIANA tree. In the course of this processing, symbol tables from separately compiled units may be read. The SEM phase completes the symbol table for a compilation unit.



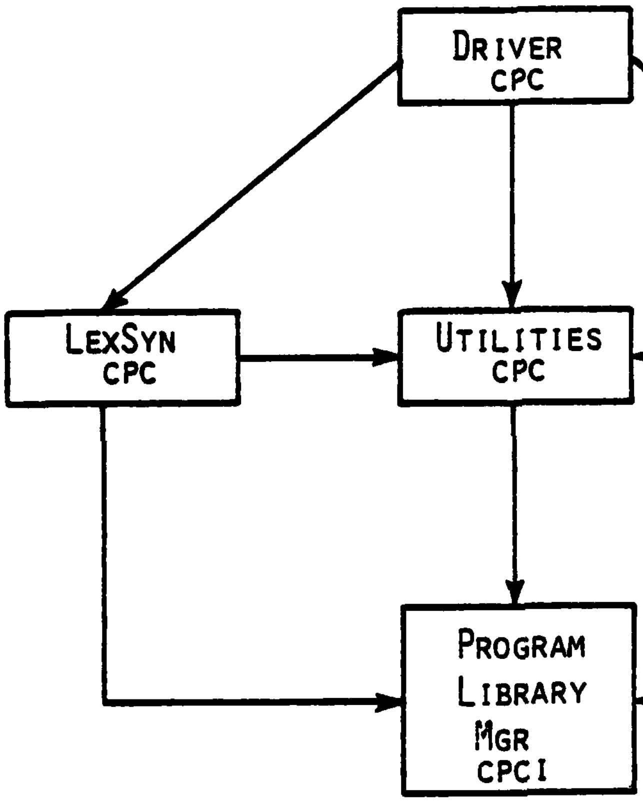
10782378-6

FIGURE 3-2: Compiler Flow of Control



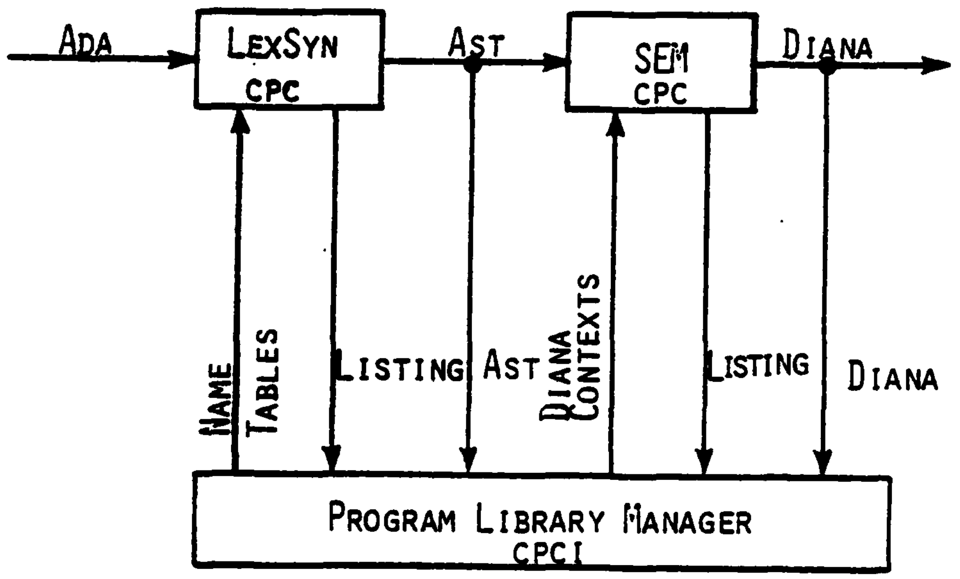
10782378-5

FIGURE 3-3: Compiler Flow of Data



10782378-4

FIGURE 3-4: Front End Flow of Control



10782378-3

FIGURE 3-5: Front End Flow of Data

Utilities needed by the Front End are bound together in UTILITIES. The UTILITIES package handles all outside interfaces. These utilities are:

(1) DIANA VMM access routines to allow paging to work.
(2) Universal arithmetic processing for static expressions.
(3) Error procedures for reporting errors.
(4) Listing procedures for outputting information for a later listing.
(5) Program library interface procedures.
(6) Query routines about DIANA attributes (e.g., is this type limited?)
(7) Statistics procedures.
(8) Tracing procedures.

3.2.5.2 Middle Part

The Middle Part consists of five components: GENINST, STATINFO, STORAGE, EXPAND, and UTILITIES.

GENINST performs generic instantiation. Each instantiation for which a body is available is replaced by an equivalent declaration, so that later phases of the compiler need not know whether a particular subprogram or package was user-supplied or generically generated.

STATINFO constructs the call graph and symbol cross references for a compilation unit, noting which references are to external compilation units. This information is used for both optimization and listing purposes.

The STORAGE phase determines the run-time representation for data of each type and the principal storage requirements for each unit (as much as can be determined statically). STORAGE generates routines for each type to carry out size determination, assignment, equality comparison, component selection, and object generation/initialization. Information computed by STORAGE is added as attributes to the symbol table portion of the DIANA tree.

The EXPAND phase carries out a major tree rewrite that removes the implicit Ada semantics and exposes address arithmetic for later optimization. Data references, subprogram and entry calls, aggregates, object creation, and Ada attributes are expanded using the routines generated by STORAGE. Checking is added to the tree when needed. A low-level tree is produced; its structure is referred to as BILL ("But It's Low-Level").

The UTILITIES package contains a driver and common routines required by the Middle Part.

3.2.5.3 Back End

The Back End consists of six components: FLOW, VCODE, TNBIND, CODEGEN, FINAL, and UTILITIES.

The FLOW phase performs machine-independent optimizations based upon machine-dependent cost criteria. The following transformations on the BILL representation are carried out: redundant constraint check elimination; constant folding and propagation; elimination of unreachable code; common subexpression elimination; code motion for loop invariants; strength reduction; and conversion of Boolean operations to transfer logic in control flow contexts.

The VCODE phase performs a tree walk simulating code generation. Instead of generating code it determines the register requirements and applicable addressing modes.

The TNBIND phase determines the location of every object that the code generator will deal with. The lifetime of each temporary name, or "TN" (variable, common subexpression, expression value) is determined. Based upon conflict information and a machine-dependent cost function, a machine-independent packing algorithm determines the register assignment for the TNs.

The CODEGEN phase uses machine-specific templates to generate a linked list of locally optimal target machine instructions. In the case of multiple potential matches, a cost function is used to determine the selection.

The FINAL phase performs machine-dependent "peephole" optimizations, and cross jumping. For the VM/370, FINAL produces segmented code; for the OS/32, FINAL performs span-dependent branch optimization. For each compilation unit, FINAL produces the input to the linker, including the "pure" storage corresponding to the unit (code and literals) and the required size for any static storage associated with the unit.

The UTILITIES package contains common routines required by the Back End.

3.3 Detailed Functional Requirements

3.3.1 Front End

3.3.1.1 DRIVER

The DRIVER is the primary user interface to the compiler. That is, an invocation of the compiler is actually an invocation of the DRIVER. The function of the DRIVER is to sequence the phases of the compiler.

3.3.1.1.1 Inputs

Input to the DRIVER is the Ada source compilation to be compiled, the program library within which this compilation is to occur, and the set of user options specified in the call to the compiler.

3.3.1.1.2 Processing

The DRIVER processes compilations by invoking the three main compiler partitions in the following order: Front End, Middle Part, and Back End. DRIVER directly invokes phases of the Front End. The Middle and Back End phases each consist of subphases that are controlled by a subdriver for that phase. Also, LISTER is a phase that is run if the user specifies the listing option (See AIE(1).PIF(1). For each phase, DRIVER performs the necessary initialization and finalization of the VMM data structures required by the phase. Placing VMM domain operations in DRIVER increases modularity by isolating VMM domain opening and closing within a single component, so that each phase is unconcerned with these functions. Also, using this design, the ability to overlay phases while maintaining VMM objects in main memory is enhanced. Such overlaying reduces VMM paging, thereby increasing the speed of the compiler.

Input to DRIVER is an entire compilation, which may consist of several compilation units. DRIVER calls LEXSYN with a compilation and the computer options as its input parameters, and LEXSYN breaks the compilation into individual compilation units, which are the units stored in the program library. DRIVER enters the abstract syntax trees (ASTs) of these compilation units into the library when they are completed by LEXSYN, i.e., DRIVER does not defer entry of the ASTs until other phases (e.g., SEM) have been run on the compilation unit. Compilation can be suspended after the generation of the AST; DRIVER later can be called to complete the compilation units from the AST stored in the library. Subsequent phases operate upon compilation units, rather than the entire compilation.

DRIVER also handles the DIANA structure needed by other phases. During processing by a phase, the data objects of the phase are maintained in VMM subdomains that are temporary KAPSE objects managed by DRIVER and separate from the program library. When phase processing is complete, DRIVER updates the program library with the results of the phase in a single indivisible operation. Thus, because the program library contains only completed objects, not partial results, abnormal terminations of the compiler (e.g., by user abort) leave the program library in a self-consistent state.

The DIANA for a unit in the program library may become outdated if a unit it depends upon is changed. DRIVER may trigger recompilations indirectly if the current compilation unit depends upon a preexisting library unit that needs recompilation. To establish the proper context for the current compilation unit, DRIVER calls the program library manager (PIF.PLTOOLS.B). The program library manager performs the analysis to determine if the library is in a consistent state for compiling the current compilation unit.

If the library is inconsistent for compilation of the current unit, recompilations are performed by the program library manager (involving recursive invocations of DRIVER) if the user has selected the automatic recompilation option (a library option). If such recompilations are unsuccessful (because of semantic errors introduced by the previous change that necessitated recompilation) the program library manager returns a cancel indication to DRIVER, so that the current compilation is terminated.

3.3.1.1.3 Outputs

The outputs of the DRIVER vary depending upon the user options. The possible outputs, for each compilation unit, are an AST, DIANA, linkable object code, listings, statistics, errors, and cross references. These outputs are entered into the program library.

3.3.1.1.4 Special Requirements

Because the DRIVER simply sequences the phases, it consumes negligible execution time in proportion to the other phases. Therefore, its execution has little effect on the speed requirements.

3.3.1.2 LEXSYN

Figure 3-6 shows the logical flow of LEXSYN in terms of functionality.

LEXSYN, the lexical analyzer and parser, reads in the source text for an Ada compilation and produces a set of abstract syntax trees, one per compilation unit. The parser is driven by a set of LR(1) tables and uses a two-level error recovery technique. The lexical analyzer, called on a token-by-token basis from the parser, is driven by a set of lexical finite-state machine tables. The lexical and parse tables are generated automatically from a regular expression notation and an LR(1) BNF grammar, respectively. The parse table generator is based upon the NYU Ada_Ed System with added logic for handling error recovery productions. See AIE(1).MGS(1) for further details.

LEXSYN also performs some preliminary semantic analysis based solely upon the content of the abstract syntax tree generated.

3.3.1.2.1 Inputs

LEXSYN has three IN parameters, the text file containing the source text for the compilation, compiler options, and the library being used.

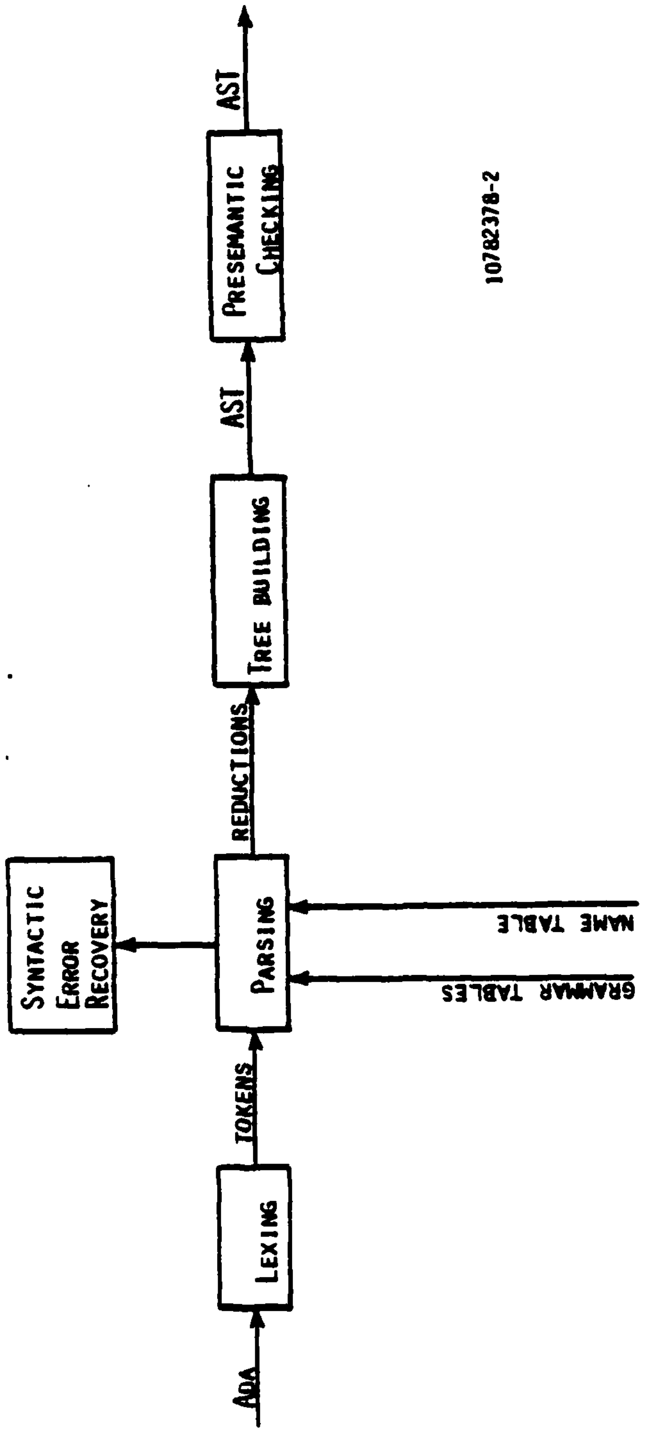


FIGURE 3-6: LEXSYN Logical Organization

3.3.1.2.2 Processing

(a) Properties of the AST. If the source text comprises a lexically and syntactically valid Ada compilation, then the abstract syntax tree is as defined in the DIANA Reference Manual (see Section 2.4) with the following additional attributes:

(1) The HAS_ERRORS attribute (a boolean) is defined as FALSE for each compilation_unit node.

(2) The MESSAGES attribute is defined for each compilation_unit node. Its value specifies the messages (i.e., errors, warnings, notes) produced by LEXSYN for the given unit. Each message is given by a tuple that identifies the exact position in the source text and the nature of the message.

(3) The SOURCE attribute is defined for the compilation node. It is a representation that allows an equivalent source text to be retrieved. This attribute permits the listing to be generated from the AST.

The SOURCE attribute will also, under user control, contain Ada comments appearing in the source, so that the full program may be reconstructed.

(4) Each node corresponding to a namescope has a SYMTAB attribute that gives the list of all labels defined in the source including statement labels and loop labels.

(5) The compiler options are saved in the AST with the compilation unit node.

If the source text contains lexical or syntactic errors, then the abstract syntax tree corresponds to the text as repaired by the error recovery algorithm. The attributes described above apply here also, except that HAS_ERRORS is TRUE for the compilation units that contain errors.

(b) Parser. LEXSYN uses a conventional bottom-up parse algorithm with one symbol look-ahead, with the distinguishing feature that a two-level error recovery technique is included. The parse stack at any point consists of PARSE_TOKENs, where each PARSE_TOKEN specifies: (1) a state in the parse table and, possibly, (2) a subtree of the tree under construction (when the state corresponds to a non-terminal symbol). The AST generated by LEXSYN differs from the derivation tree in that the AST does not depend on the details of the particular LR(1) grammar which is used; e.g., "singleton" rules of the form <non-terminal symbol$_1$> ::= <non-terminal symbol$_2$> are not present in the AST. The AST is constructed by the parse actions associated with the rules of the grammar.

The parser begins by creating a name table with one entry for each unique identifier seen within a compilation unit (excluding variations in capitalization). In the event the compilation unit is a body or subunit, the parser first copies the name table of the associated higher library unit, and then continues with new names occurring within the current compilation unit. Each name table entry has an associated DIANA node pointer to the most recent declaration of such an identifier. In many cases, this pointer is null, and will be filled in by semantics processing. Package STANDARD identifiers are an exception.

For package STANDARD, its name table is not copied into the name space of each compilation unit. Instead, when the parser performs a hashed lookup of a token to see if it is a keyword, it simultaneously sees if the token is a STANDARD identifier. If so, the DIANA node pointer for the STANDARD item is inserted along with the identifier name in the table. Thus, only STANDARD identifiers actually referenced by a compilation unit or its lexical predecessors will appear in its name table.

The syntactic error recovery technique is a two-level process. At the point of detection, a local repair is attempted based on the following alternatives:

(1) deletion of current input token;

(2) insertion of a legal shift symbol before the current input token;

(3) replacement of the current input token with a legitimate shift symbol (e.g., spelling correction).

The cost of each repair is computed by scanning ahead a fixed number of tokens and running the parser to see whether further errors are introduced. If one of the repairs is sufficiently economical, it is carried out. Otherwise, a secondary approach is taken, which consists of the following steps:

(1) Pop the stack until the top state has a shift transition for the special <u>error</u> terminal symbol (the grammar has been augmented with rules which end with this symbol).

(2) The action routine for each of these <u>error</u> rules advances the input until either: (a) a legal shift symbol is found, or (b) a special "beacon" symbol, such as a semi-colon, is found.

In the first case, parsing resumes simply by reading the symbol. In the second case, the parse stack is popped until a state is uncovered with a non-terminal transition to a state from which the beacon symbol may be read. This state is pushed onto the stack and parsing resumes.

At the point of error detection, a message tuple is built and appended to the value of the MESSAGES variable. When the compilation_unit node is eventually produced, the MESSAGES attribute will be set to the value of this variable.

Use of a complex error recovery scheme involves a small overhead for programs which have no syntax errors. In order for the error recovery scheme to work, tokens and reductions are buffered. The cost of this buffering in programs without syntactic errors is estimated to be 2% of the total parsing cost (measured in a SETL implementation of the parser).

(c) Lexical Analyzer. The lexical analyzer, invoked by the parser to produce the next input token, is a finite-state machine simulator. The token which it produces consists of a token class (a terminal symbol in the grammar) and, depending on the token class, a token value (a pointer to the character string comprising the token).

Reserved words are detected by a perfect hash function which is given an identifier; thus, the individual character transitions do not have to be built into the finite-state machine tables. Each occurrence of a letter in an identifier is normalized by a lower-case to upper-case conversion. Hash tables (one per compilation unit) are maintained for pointers to nonreserved words and literals. Thus, different occurrences of the same identifier or literal map to different tokens with the same token value.

Processing of "." and "'" are handled by the LEXSYN phase. For ".", the problem is to distinguish 1.0 from 1..2. This is done using an extra character lookahead when processing numbers.

For "'", the problem is to distinguish attribute selection from type qualification from character literals. The parser keeps track of the last token seen. If the token was an identifier, then the next construct cannot be a character literal. Whether it is attribute selection or type qualification is resolved by semantics. If the previous token was not an identifier, then the parser processes assuming a character literal.

The LIST pragma is processed by the lexical analyzer. An appropriate output for the SOURCE attribute is produced, to reflect where the listing is to be turned on or off.

(d) Presemantic Analysis. While the abstract syntax tree is being created, some semantic checks are performed. These depend only upon the syntax of the current compilation unit, and do not require access to symbol information. Errors detected will generate semantic error messages, and cause a flag to be stored in the tree.

The benefits of presemantic checking are: errors reported sooner, recompilation time from the AST is improved, and code is moved out of the larger phase (SEM), giving more space for paging data.

The checks performed are:

(1) Check that pragmas only appear at the following places in a program:

  (a) where a statement would be allowed
  (b) where a construct whose name ends with 'declaration' or 'clause' would be allowed
  (c) following a construct that ends with a semicolon
  (d) before a reserved word when but not within an exit statement
  (e) where a compilation unit would be allowed

(2) Check that enumeration literals in a type declaration are not repeated.

(3) Check that if a private or incomplete type declaration is given, then the private part completes it.

(4) Check that labels, loop identifiers and block identifiers that occur are "named uniquely", for the enclosing body of a subprogram, package, or task.

(5) Check that subprogram end designated matches subprogram name.

(6) Check that loop id matches at beginning and end of loop.

(7) Check that ending label to accept statement matches entry name.

(8) Check that block simple_name matches at beginning and end of a block.

(9) Check that, if a loop name is specified, then the EXIT statement occurs within the loop named.

(10) Check that, if a loop name is not specified, then the EXIT statement occurs within a loop.

(11) Check that EXIT does not leave subprogram body, package body, task body or an accept statement.

(12) Check that a RETURN statement occurs only within a function body, a procedure body or an accept statement.

(13) Check that a RETURN statement for an accept statement, procedure, or package body does not include an expression.

(14) Check that RETURN does not transfer control out of a package body, or task body.

(15) Check that a GOTO statement does not go from outside into a command statement or exception handler. A GOTO statement must not go from one of the sequence of statements of an if, case, or select statement to another.

(16) Check that a GOTO statement does not go from an exception handler to another, nor back to the statements of the corresponding block, subprogram body, package body, or task body.

(17) Check that a GOTO statement does not transfer control out of a subprogram body, package body, task body, or an accept statement.
A GOTO statement must not transfer control from outside to inside the body of a subprogram, program, or task.

(18) Check that only a formal parameter with IN mode has initializations.

(19) Check that if both positional and named associations are used in the same call, positional associations occur first, at their normal position: once a named association is used, the rest of the call must use only named associations.

(20) Check that FUNCTION parameters are all of mode IN.

(21) Check that a function body has within it, a return statement with an expression.

(22) Check that end identifier of package specification and package body, as well as package body identifier, matches package specification identifier.

(23) Check that the end identifier of task specification and task body, as well as task body identifier, matches task specification identifier.

(24) Check that an ACCEPT statement is inside a task body, and corresponds to an entry declaration in that task.

(25) Check that an ACCEPT statement is not within a subprogram, package or task unit which is within that task.

(26) Check that a SELECT statement is within a task body.

(27) Check that at least one select alternative starts with an ACCEPT.

(28) Check that if a terminate alternative is given, a delay alternative is not also specified.

(29) Check that if a terminate alternative or a delay alternative is given, an else part is not also specified.

(30) Check that at most one terminate alternative is allowed.

(31) Check that at least one task name is given for a select.

(32) Check that OTHERS is last choice in exception handler, if it occurs, and has no other names with it.

(33) Check that OTHERS is the last choice in case statement, if it occurs, and has no other names with it.

(34) Check that if a RAISE with no name appears, it is within a handler, and not within nested subprogram body, package body, task body.

(35) Check that if a code statement is in the sequence of statements of a procedure body, all other statements in this procedure body are code statements.

(36) Check for wrong declaration order: statements not followed by declarative items.

(37) Check that GOTO uses a valid label name.

3.3.1.2.3 Outputs

LEXSYN has one OUT parameter, the abstract syntax tree list, abbreviated AST. *LEXSYN* may also generate error messages (see Appendix A).

3.3.1.2.4 Special Requirements

In order to meet the overall compiler speed requirements, the LEXSYN phase should run at 6000 statements/CPU-minute. To accomplish this, it may be necessary to write out a linear intermediate language, since there is a relatively high overhead associated with creating and paging tree nodes a VMM subdomain. Since the SEM phase rewrites a whole new tree, filling in semantic attributes, the current design calls for two tree creations in the Front End. To reduce the cost, LEXSYN may write out a compressed structure which is not a DIANA tree, using VMM. The actual tree would then only be created once, by the SEM phase. It would be possible, however, to create the normal AST from the output of the LEXSYN phase using a special tool.

Another alternative is for the compiler to be able to switch phases without VMM having to close and reopen the subdomains and domain. The SEM phase would be overlaid on the LEXSYN phase, and have access to the VMM objects, which would still be resident in the paging buffers in most cases.

The exact implementation chosen will depend upon performance measurements and the overall speed requirement of the Front End.

3.3.1.3 SEM

Figure 3-7 shows the logical processing done by SEM.

The SEM phase derives a representation of the static meaning of the compilation unit and performs the static semantic checking required for Ada. This phase maps an occurrence of a designator or character literal to its definition, resolves overloading in the case that the name itself is insufficient, and maintains direct and USE visibility of identifiers. The SEM phase is responsible for completing the symbol table built by the LEXSYN phase.

3.3.1.3.1 Inputs

The SEM phase has one IN parameter. The IN parameter is the abstract syntax tree for a compilation unit, as built by the LEXSYN phase. This includes the partially completed symbol table and the name table.

3.3.1.3.2 Processing

(a) General Strategy. The SEM phase expands the input tree, initializing values for DIANA attributes that are not present in the abstract syntax tree. This process involves creating a new VMM subdomain for the DIANA tree for this compilation unit. As SEM walks the AST, the DIANA tree is built up.

The semantic analysis portion of this phase is implemented by a recursive tree walker that visits all the nodes on the tree in prefix (top-down) order. The walker is organized so that parts of the tree can be replaced by new tree nodes in the few cases that actual modifications to the tree are necessary. The remainder of this phase consists of a set of procedures that implement the symbol table management strategies and a set of mutually recursive procedures that do the processing required for each of the node types in the abstract syntax tree.

The tree walker invokes the overloading resolution procedure in expressions. Overload resolution uses a three-pass tree walk to derive the correct definition of an identifier or operator that does not have a unique definition. (See section (b), Meaning Resolution Strategy.)

(b) Meaning Resolution. The Front End of the compiler has two basic tasks to perform: determining the meaning of the program, and enforcing restrictions about the use of Ada constructs. Determining the meaning of the program consists of name, type and "construct"

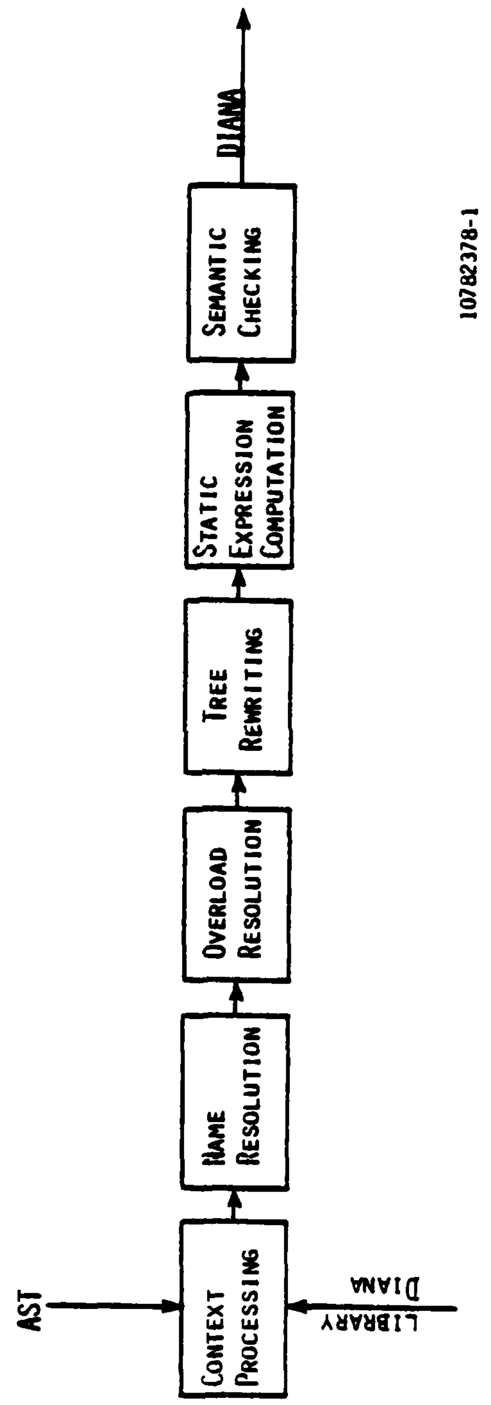


FIGURE 3-7: SEM Logical Organization

resolution. Name resolution means identifying the explicitly or implicitly declared entity associated with each identifier. Type resolution means identifying the type of every name and expression in the program. "Construct" resolution means identifying the particular semantic construct intended when there are several possible interpretations of some syntactic construct (e.g., NAME(1) might mean a function call, an array reference, etc.).

Determining the program's meaning, is done in two passes over the tree. These passes are labelled pass 2 and pass 3, with pass 1 being construction of the tree.

Pass 1 creates the tree (done by the LEXSYN phase). Pass 2 is a bottom-up walk that propagates sets of choices of what the node might mean. At each node, the meanings available at the node are matched with the arguments to the node, and only valid combinations of meanings and arguments are preserved. For a meaning and its arguments to match, the number and types of the arguments must match, and any named associations (either user-specified or language-specified) must be correct.

Thus pruning of meaning is happening during this bottom-up pass. At the top of the tree, in a valid program, a single unambiguous, consistent choice is found. In invalid programs, either no choice is found, or more than one is found. Assuming a single choice is found at the top, pass 3 then goes down again, and finishes pruning in the lower levels, now that higher levels are unambiguously known. By the end of pass 3, all nodes of the tree have their unique type and symbol being referenced associated with them.

The general process of resolution of a node consists of:

(1) For terminal nodes

For simple names - Call LOOKUP to determine the symbols which are legal interpretations. Each interpretation gets a separate entry on the CHOICES list.

For literals - A CHOICES list is created for the literal, and filled in with legal type symbols.

(2) For non-terminal nodes take the set of CHOICES provided on the operator sub-node and for each choice, process the argument sub-nodes' information to determine if the choice is legal. For each operator choice, this may result in zero, one, or many possible choices to be entered in the CHOICES list for the non-terminal node.

This is done for all nodes in the subtree, bottom up, until the top node is completed. At that point, the meaning will be: unambiguous, null, or ambiguous. If the meaning is null, the program is in error. If the meaning is ambiguous, then some language rules may require that the compiler pick a meaning; i.e., provide a tie-break criterion. This is particularly true with expressions involving numeric literals, where the compiler may have to say "If no specific user type is clearly correct, make the answer of type UNIVERSAL x". Other tie-break rules may require the compiler to select INTEGER, and so on. This is language construct specific. The important point is that sometimes the compiler has additional information available to disambiguate multiple meanings, and sometimes it doesn't, in which case once again the program is in error.

There are gaps in the Ada language definition with regard to relationals which return Boolean as their type. If "..." <"..." is seen, the output result type is known to be Boolean and the input argument types are known to be arrays, but exactly what type is not known. Processing will depend on the choice made regarding the validity of a statement, if only one array type is currently defined. If the program is valid at the top, then the unique meaning and type is known. The compiler then goes down the tree, knowing the type of the node because the higher node knew what type it was. Knowing the type of a node, the correct choice entry is selected as the unique meaning of the node. This is done until the bottom of the tree, at all terminal nodes, at which point type and name resolution are complete.

(Actually, since the contents of an aggregate have not yet been seen, the aggregate is treated as a whole new subtree to be processed underneath us, where we know the type in advance, and thus know the types of each component. Thus, the subtree starts in pass 3).

On the third pass, during the ascent of the tree, the DIANA node is transformed to that appropriate for the resolved meaning, thus completing construct resolution.

Use of Wild Card Types. For literals, instead of computing the set of all visible types which this literal might be, a "wild card" type mark is passed indicating what class of types it might be (i.e., "wild card array", "wild card integer", etc). Since the routines which prune sets of choices use intersections of information, those routines are built to correctly handle the wild card type intersected with any other type, including more general wild card types (e.g., intersecting "wild card real" with "wild card float" will yield "wild card float"). Naturally, a wild card type will eventually intersect into some specific type in a correct program.

There is a wild card type for: discrete, scalar, numeric, integer, composite, real, float, array, 1-dimension array, boolean, access, any-type, non-limited-type, and record. There is no need to have a wild card type for 1-dimensional array of boolean, or 1-dimensional array of character, or the like, because the component type of the array will be stored in a separate field of the choice entry to be described under the wild card builtin symbol.

The constructs marked initially with wild card types are:

integer literals	=>	wild card integer
real literals	=>	wild card real
NULL	=>	wild card access (of any-type)
NEW X[(...)]	=>	wild card access (of X)
aggregate	=>	wild card composite (of any-type)
"......"	=>	wild card 1-dimensional array (of CHARACTER)

Use of Wild Card Builtin Symbols. Instead of returning a list of all legal symbols when calling lookup on a simple name which is an operator symbol (builtin), a special "wild card symbol" is used to represent predefined and derive-inherited predefined operations. The set of symbols returned from a lookup thus has all user-defined symbols, and one wild card symbol to represent all possible legal predefined meanings of an operator.

The following symbols are treated generically:

+ - ABS * / REM MOD ** AND OR XOR = <> & >= <= NOT /=

Note: since the user may never explicitly define "/=", the compiler will automatically generate a "/=" definition whenever the user redefines "=".

The wild card symbol will have information associated with it which will be interpreted by special choice-pruning routines. These routines will interpret the arguments subnodes CHOICES lists in light of the requirements of the wild card symbol. To support wild card builtin processing, each entry in the CHOICES lists will need additional information.

Each choice entry already had:

(1) the specific result type of the choice

(2) the specific DEF_ID for an identifier

These fields are filled in when the information becomes known.

Additional information is added to the CHOICE entry, which applies when the entry is wild card or has wild card sub-nodes. This information, needed for correct pruning, is:

(1) a flag marking this as a wild card builtin

(2) restrictions on the result type (e.g., "wild card integer")

(3) restrictions on the visibility (e.g., "must be in P")

(4) restrictions on components (e.g., for arrays, the component type must be x, and its visibility must be y)

Visibility information is needed because Ada allows the user to write "P.+". Were there no selected operators allowed, visibility information would not be required.

The result type restriction is an enumeration of:

ordinary	- use of specific result type of the choice
wild card	- one of the wild card types

The builtin flag is a boolean true or false.

The visibility restriction is a list of region pointers of:

reachable-region	- a fake region meaning unlimited access allowed, used for types which may be hidden, but available. All literals have this visibility.
visible-region	- a fake region meaning must be directly visible. All names which are not dot-selected have this visibility.
specific region	- used to allow a visible-spec region, private-spec region or a body region of a specific package. Dot-selected operators have this visibility.

The region attribute is filled in for simple name nodes of the wild card operator by looking at the visibility of the type causing the "implicit" declaration of this wild card symbol. If dot-selection was used to reference this builtin, then specific regions must be named. Otherwise, visible-region is the appropriate visibility. If the types of the arguments are NOT the same (i.e., mixed arithmetic like "*"), then the visibility is the intersection of each argument's visibility. This may be null, in which case the program is illegal.

The component restriction is a record which handles information about array components and access components. The record is:

Result type restriction	- same enumeral as above, refers to the component type
Specific result type	- type of the component if known
Visibility restriction	- same as earlier, for the component type
Component restriction	- recursive ptr to comp_restrict record type, for arrays of arrays of ..., and access of X

Processing For Terminal Nodes. If the node is a literal, store its appropriate wild card type on its choice entry, and add reachable as its visibility restriction.

If the node is a dot-selected operator, store the package regions corresponding to the dotted name as its visibility restriction.

If the node is some other name, store directly visible as the visibility restriction, and the type of each choice entry is that of the DEF_ID being considered.

Processing For Non-Terminal Nodes. If entry is not wild card builtin: For non-wild card entries, argument entries must match the types expected by the symbol entry. If no combination of argument entries matches, then the symbol is rejected as a choice. If a combination is found, then the symbol becomes a choice, and further examination of the arguments for this entry may cease (you cannot get duplications arising). In the event the argument's type is some wild card type, that will be acceptable in matching providing the wild card type is compatible with the expected type for that argument. Notice that only arguments may have wild card types at the moment, because this is not a wild card builtin symbol choice.

If entry is generic builtin: For wild card builtins, argument entries must match types, keywords, visibility restrictions, and component restrictions.

Type matching is performed by special routines, which can handle the use special markers on a wild card symbol. The markers provide the normal wild card type restrictions on the arguments, but they add additional restrictions on the relationships of arguments and result type. For example, for the ">" operator, the arguments may be of any scalar type, but they must match each other. The resulting type is always BOOLEAN. How these relationships are represented is managed by the pointed-to symbol and the special routines which match wild card symbols and their argument lists.

Visibility matching is done by intersecting the visibility restrictions of the arguments and the operator. The match means finding the same pointers in both sets, except that:

(1) The reachable fake region matches any region and returns the other region.

(2) The visible fake region matches a region only if the other region is on the lexical visible stack or the use visible stack, or is a visible fake. The result is the region matched.

(3) Other regions match only if the region pointers are the same.

Component matching is done by matching the contents of the record, throughout all levels provided by the recursive pointer. Content matching is done as described above for each type of content.

For an example of wild card symbol processing, consider the Ada fragment:

```
    package P is
    type t is (aa, bb, cc);
    v: t;
    and P;
if P.v. < P.bb then ...
```

The if statement has no direct visibility over the interior of the package, so the "<" is not visible. In processing the tree, the nodes for P.v and P.bb would have visibility restrictions which would require visibility over the package spec for P. The restrictions stem from the fact that the type used is defined in P. The wild card symbol "<" would have a directly-visible restriction. These restrictions are incompatible, because declarations within P are not directly visible. Therefore no meaning of "<" would be found and the program considered in error.

(c) <u>Symbol Table Design and Separate Compilation</u>. The symbol table is a permanent DIANA data structure. There is one symbol table per compilation unit. The table is created by the parser, and augmented by the semantics phase. Lookups are done using information about lexical visibility, and information about USE visibility.

Any declaration occurs within the scope of a particular region, and is given a sequence number reflecting where in that region it is declared.

A lookup procedure is provided which returns a list of the possible DIANA nodes that an identifier may currently be. This list is generated by lexical visibility, the overloading and hiding rules of Ada, and possibly by USE visibility as well. The actual implementation of lookup involves caching answers from previous lookups and caching lookups for USE visibility. The details of caching are unimportant at this level.

For lexical visibility, the compiler keeps a stack of currently visible regions and currently visible sequence ranges. Whether a DIANA declaration node is visible can be determined by seeing if the region in which it was declared is on the region stack, and whether the sequence number of the node is within the current active sequence range of the region.

The current sequence number is needed when attempting to reestablish correct visibility of subunits. In such cases, additional declarations may have shown up in the region containing the stub, <u>after</u> the stub was declared. These additional declarations are not visible to the separately compiled subunit, and this is managed using the sequence number.

For USE visibility, the compiler keeps a stack of currently USEd units. If lexical lookup is not sufficient, either because no entry was found, or because only overloadable entries were found, then use lookup is added. In essence, each current USE-visible region on the use stack has its name table examined for the appropriate identifier entry and its SAME_NAME chain is used to consider additional declaration candidates, subject to overloading and hiding rules of Ada.

When SEMANTICS begins processing a compilation unit, it first establishes the context of that unit. This context starts with the higher lexical scopes still active, and symbols thus visible. If the current compilation unit is not a top-level specification, the symbol table of the immediately higher lexical scope is read in and its DIANA node pointers are transferred into the current name table.

When SEMANTICS sees a WITH clause in the beginning of the current unit, the entry for the unit named is located in that unit's symbol table and the DIANA pointer transferred into the current compilation unit's table. This makes the identifier visible in the current symbol table, and, if the identifier happens to overload a STANDARD identifier, insures that this information is not lost. It is not lost, because the WITH'ed symbol table entry has the unit's DIANA node pointer followed by a SAME_NAME link to the STANDARD DIANA node.

When SEMANTICS processes a declaration, the DIANA node corresponding to the declaration will be inserted in the symbol table with the the appropriate entry in front of all prior declarations. This insures that lookup sees the most recent declaration first, and that prior DIANA nodes are not re-written upon. This is important in separate compilation to insure that separately compiled units are accessible in a READ-only fashion. The DIANA node is also assigned a sequence number reflecting when it was declared within a region.

Lexical visibility is then a simple matter of looking up the identifier's text name via hashing into the symbol table and retrieving the corresponding DIANA node pointer. This node is the head of a SAME_NAME chain of all DIANA nodes from declaration of similarly named items.

Not all items on the SAME_NAME chain are currently visible. The combination of the stack of currently visible regions and current sequence numbers valid within a region enable lookup to determine which SAME_NAME entries are currently visible by lexical visibility.

USE visibility merely extends lookup to see SAME_NAME chains from symbol tables corresponding to the compilation units in which the USE'd names arose.

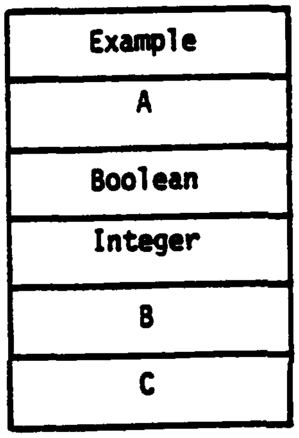
The following examples illustrate the operation of the symbol table on simple program units. Figure 3-8 shows an example package and the resulting name table after parsing, prior to semantics. Note that although package Example declares two functions with the identifier A, the name table has only a single entry for the identifier A. This situation illustrates the property of the name table that each unique identifier appears only once in the name table, regardless of the number of times the identifier is used in declarations. Semantic analysis augments this structure with definitions for each declaration. The advantages of this property for overload resolution are discussed below. If the user selects to perform only parsing, LEXSYN stores the name table as shown in Figure 3-8 in the program library, along with the AST.

Figure 3-9 shows the name table for the same package during semantic analysis. The name table has been augmented with the DIANA DEF_ID nodes that are the corresponding definitions for the identifier. The name table defines a mapping between identifiers and DIANA DEF_ID nodes contained within the DIANA tree. This mapping makes it simple to retrieve a DIANA DEF_ID node, given a DIANA UsedId node (which contains the lexical symbol representation). Note that the entry for A contains two DEF_IDs, one for each of the (overloaded) functions defined within package Example. A lookup of the identifier A returns the list of possible definitions for A so that, when analyzing an expression using A, overload resolution can select the appropriate definition (or announce an error).

The structure shown in Figure 3-9 along with additional information for optimizing lookups, is built during semantic analysis. When semantics completes, a portion of this structure is saved along with the DIANA in the program library. The saved portion contains the mapping between the identifiers appearing in the compilation unit and the DEF_IDs defined in this compilation unit. To avoid redundant information, the saved portion omits the DEF_IDs defined in other compilation units (e.g., the type ids for Boolean and Integer in package Example) and the information computed for optimized lookups.

Because the DEF_ID nodes for all identifiers that appear in a compilation unit are placed in a single name table, additional information is necessary to indicate lexical nesting. This additional information is a region identifier associated with each region (cf. LRM 8.1) in the source program, as well as a lexical visibility stack which dynamically reflects the lexical visibility during semantic analysis. Each DEF_ID defined in a region contains the corresponding region identifier. The lexical visibility stack contains the stack of lexically open regions. (See Figure 3-10).

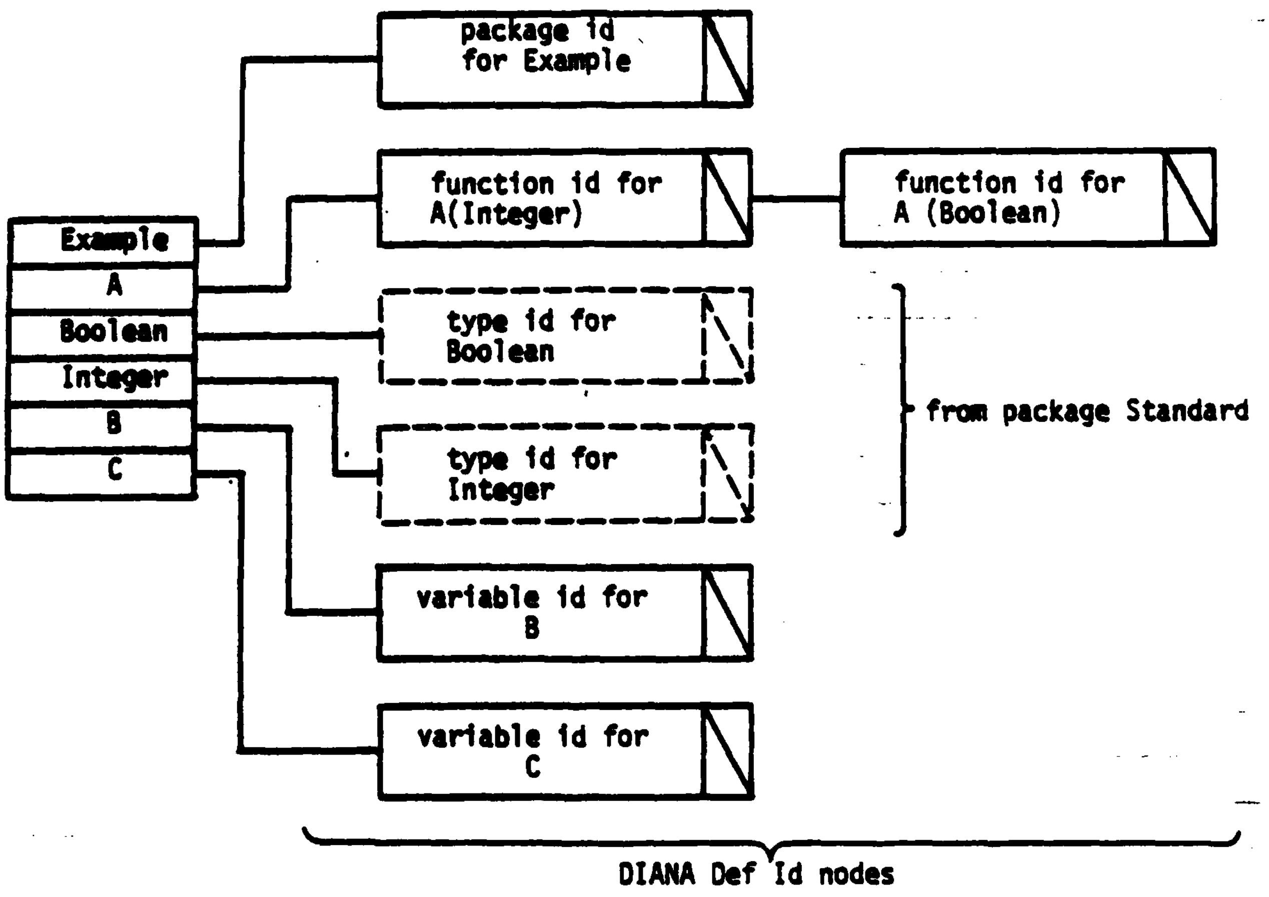
```
package Example is

     function A return Boolean;
     function A return Integer;

         B: Integer;
         C: Boolean;

end Example;
```



10782378-10

FIGURE 3-8: Name Table After Parsing

```
package Example is:
     function A return Boolean;
     function A return Integer;
     B :  Integer;
     C :  Boolean;
end Example;
```



10782378-9

FIGURE 3-9: Name Table After Semantics

Dashed boxes indicate identifiers defined in other packages. In this example, the definitions for Boolean and Integer are package standard.

In looking up the possible meanings of an identifier, the lookup routine considers only identifiers having a region id currently on the lexical visibility stack. It ignores DEF_IDs having a region id other than those currently on the stack. Because DEF_IDs are inserted at the beginning of the name table list, the lookup routine finds the most recent definitions first and rejects those whose region id is greater than the region id currently on the top of the lexical visibility stack. Lookup terminates when a nonover loadable DEF_ID is encountered (because a nonoverloadable hides any DEF_ID later in this list), returning a list of possible DEF_IDs.

Thus, the ordering of DEF_IDs in the name table allows lookup to be efficient. This organization also allows easy detection of illegal redeclarations; before a new DEF_ID is inserted, the list is checked for a homograph (Ada LRM 8.4) in the region on the top of the stack. Note that the lexical visibility stack is needed only during semantic analysis and is not saved in the program library.

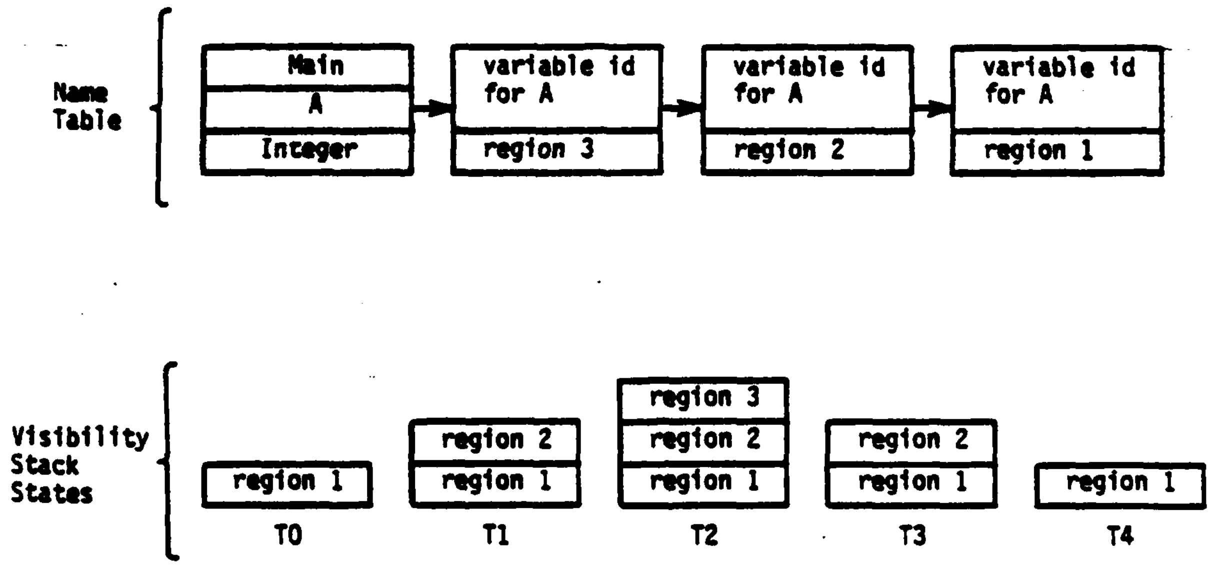
Additional structure is necessary for visibility rules concerning "USE" clauses. If a DEF_ID is not found using the immediate scope lookup, the identifiers that are "USE" visible are inspected next. The packages currently "USE" visible are maintained on the USE visibility stack. When a USE clause is encountered, an entry for the package is pushed on the USE visibility stack. When the scope of a USE clause is left, the USE visibility stack is popped to eliminate USE visibility for the corresponding package.

During USE visibility lookup, DEF_IDs from USE visible packages are entered into the name table for the current compilation, along with an indication of the defining package. In constructing this list, Ada visibility rules are checked. These rules specify that if more than one USE visible symbols for the same name occurs, then they must be subprograms or enumeration literals. That is, the occurrence of any nonoverloadable symbol cancels USE visibility unless it is the only symbol found.

Figure 3-11 shows a simple main procedure that uses the package Example (from Figure 3-8). When the "USE Example" clause is encountered, Example is pushed on the USE visibility stack. The name table entry within Main for Example refers to the DEF_ID of Example. In the assignment to B, when immediate visibility lookup for A fails, the USE visible DEF_IDs for A are looked up. This lookup begins by looking in the Example name table for the identifier A, which locates the list of DEF_IDs for A. Because the only DEF_IDs found are for subprograms, this list is returned as the result of lookup for A (and can then be used in overload resolution for the assignment statement).

(e) <u>Aggregate Type Identification</u>. Ada language rules require that the type of an aggregate be known from its context. The compiler does not have to examine the aggregate to determine its type. Once the type has been assigned to the aggregate, the types of each of its components are also known, and overload resolution may proceed for each component with minimal work.

```
procedure Main is  -- region 1
   A : Integer;
begin
   -- lexical visibility at T0
   declare         -- region 2
      A : Integer;
   begin
      -- lexical visibility at T1
      declare      -- region 3
         A : Integer;
      begin
         -- lexical visibility at T2
      end;
      -- lexical visiblity at T3
   end;
   -- lexical visiblity at T4
end Main;
```



10782378-11

FIGURE 3-10: Lexical Visibility Stack

The lexical visibility stack indicates the lexically open scopes during semantic analysis. Use of identifier A refers to the DEF_ID corresponding to the region highest on the stack.

```
with Example;
procedure Main is
   B : Boolean;
   use Example;
begin
   B := A;    -- assignment to local B
end Main;
```

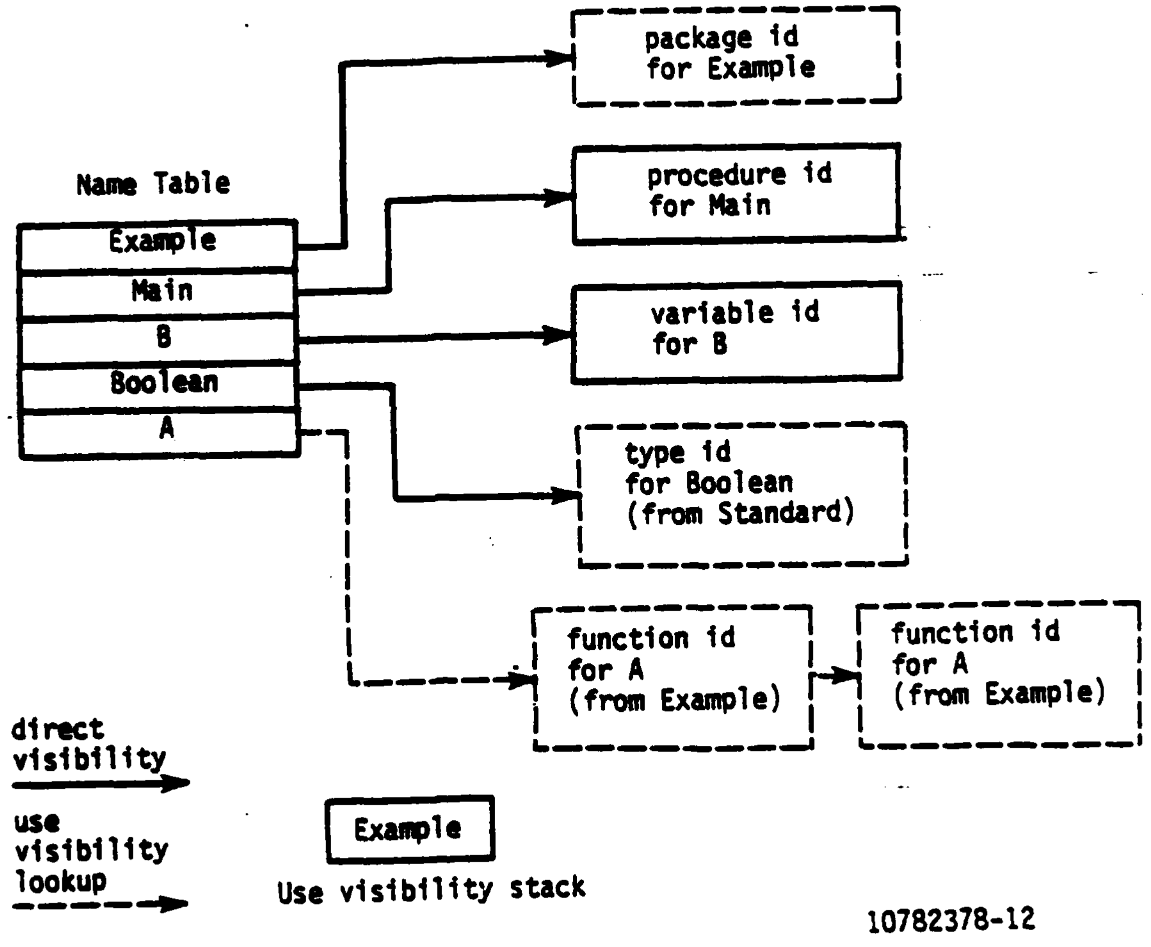


FIGURE 3-11: USE Visibility

If a symbol is not found using ordinary visibility lookup, "USE" visible lookup is performed. The USE visibility stack indicates the packages current "USE" visible. Dashed boxes indicate nodes defined in other packages.

(f) Apply Operator and Dot Selection Operator. The abstract syntax tree contains an apply node in the case that a name is followed by a parenthesized list of arguments, or an Ada operator is specified. The Ada "dot" operator likewise can mean either name qualification or record component selection. Semantic analysis changes these to the appropriate DIANA representation. Analysis in both cases determines the possible set of meanings for the name portion of the tree, and then uses this information to drive overloading resolution of the expression portions of the tree. For the case of dot selection, the names must always be unique, whereas for the apply node the names can be overloaded and must go through full overload resolution.

(g) Compile Time Arithmetric. Ada language rules require that the compiler evaluate arbitrarily accurate arithmetic between named numbers in the user program (see [Ada LRM, 4.10]).

The semantic analyzer uses the Universal Arithmetic package provided by the Front End.

(h) Derived Types. A derived type inherits the operations of its parent type. Symbol table entries for an abbreviated form of the inherited operations are created but the subprogram bodies will not be copied. For built-in functions, separate symbol table entries will not be kept, but rather, a generic builtin symbol for a particular operation will be used, and the overload resolution algorithm will be modified to deal with it. See (c) above.

(i) Pragmas. SEM accepts a variety of pragmas. The set of pragmas accepted is shown below. Most of the pragmas are defined by the Ada language, but a few have been created to assist code generation and run-time heap management.

Language-defined		AIE-defined
CONTROLLED	ELABORATE	MARK_RELEASE
INTERFACE	INLINE	MONITOR
LIST	MEMORY_SIZE	STATIC
OPTIMIZE	PACK	
PAGE	PRIORITY	
STORAGE_UNIT	SUPPRESS	
SYSTEM_NAME		

The MARK_RELEASE, MONITOR, and STATIC pragmas are described in AIE(1).KAPSE(1), 3.3.2.4.2.8, and 3.3.2.4.2.1.

The INTERFACE pragma is described in AIE(1).PIF(1).

The remaining language-defined pragmas are defined in the Ada LRM.

(j) Attributes. The Front End recognizes all of the attributes defined in the Ada language, and handles validating the context and arguments supplied. Those attributes which have static values in Ada, are handled by the static expression code of the compiler.

3.3.1.3.2.1 Generics

Generics processing occurs as part of semantic analysis. Semantic analysis of generics has three major functions: (1) to ensure conformance of generics usage with Ada rules; (2) to create a representation of generic instantiations that is convenient for processing by later compiler phases; and (3) to facilitate sharing of generic bodies across multiple instantiations. Each of these functions is described below.

3.3.1.3.2.1.1 Generics Semantic Analysis

Semantic analysis of generics occurs at three points: the generic declaration, the generic body, and the generic instantiation. The generic declaration establishes the properties of the formal generic parameters. Semantic analysis completes the DIANA representation of generic formals and checks that the formal parameter declarations are consistent with Ada rules. Examples of such checks are (1) ensuring that only formal objects of mode IN have default expressions, (2) ensuring that the only form of discrete range in a generic formal constrained array type is a type mark, and (3) ensuring that discriminants of generic formal private types do not include a default expression.

The corresponding generic body establishes the template to be used by instantiations of the generic. Semantic analysis checks the semantics of the generic body and produces the DIANA representation of the body template. Since the correctness of a generic instantiation in general can depend upon the characteristics of the generic body template, semantic analysis of the body alone cannot decide the correctness of all possible instantiations. That is, the instantiation of a semantically correct generic body may be illegal, depending upon the generic actual parameters. For example, an unconstrained array type passed to a generic formal private type results in an illegal instantiation if the body declares objects of the formal private type. To simplify checking of instantiations and to support diagnostics, a list of actual parameter dependencies and the points within the generic body corresponding to these dependencies are associated with the DIANA representation of the body.

Generic instantiation results in matching of the generic actual parameters with the generic formal parameters in the generic declaration, producing the DIANA representation of the generic instantiation. This representation is not a full expansion of the template with the generic actuals. Instead, it contains the attributes of the generic actuals parameters and refers to the corresponding template (if the template is available). The instantiation is processed based only upon the information available from the generic declaration, without requiring that the body be available. If the generic body is in the same declarative list, the new specification refers to the corresponding template.

Recursive generic instantiation, either directly or indirectly, is checked for and prohibited. To enable checking for recursive instantiations, semantics maintains a dependence graph, associated with the program library. Nodes in the graph represent generic units and arcs represent instantiations. An arc from node A to node B indicates that A instantiates B. Checking for recursive instantiation, which occurs upon encountering an instantiation, involves traversing this graph to detect cycles. A cycle indicates a circular dependence, i.e., a recursive instantiation. Arcs in the graph connect to both the specification and the body of the instantiated generic, because the dependence is upon both the specification and the body. This method enables checking for mutually recursive generic instantiations in separately compiled generic bodies.

In checking instantiations, the instantiation actual parameters are matched with the corresponding generic formal parameters given in the generic declaration. This checking ensures that generic formal parameters receive appropriate actual parameters in the instantiation, e.g., that a type formal parameter receives an appropriate actual type and that a subprogram formal parameter receives an appropriate actual subprogram. For generic objects of mode IN OUT, a check is made that the variable is not a dependent subcomponent of an unconstrained variable. Also, objects of mode in are checked to ensure that the actuals are not a limited type.

For formal private types, the correctness of the instantiation may depend upon the generic actual parameters and the characteristics of the body. For example, if the body declares objects of a formal private type, the instantiation is incorrect if the actual type supplied for the formal private type is an unconstrained array type. As discussed above, semantic analysis of the body records the points at which the correctness of the instantiation depends upon the characteristics of the body and the actual parameters. Given this list of dependencies, the instantiation simply must check the list to determine if the body and the actual parameters are incompatible. This approach simplifies generic instantiation by avoiding the requirements to semantically analyze the entire program unit obtained after the expansion of the generic template.

For formal array types, checks are performed to ensure that the actual and formal parameters have the same number of index positions, that they are either both constrained or both unconstrained, and that they have the same index types. For formal access type, checks are performed to ensure that the actual and formal parameters designate the same type of object. For formal subprograms, SEM checks that the corresponding implicit renaming declaration is legal.

Because of the possibility of generic subunits, the generic template in general may be unavailable at the point of an instantiation. The instantiation has available the generic declaration to perform the matching of actual and formal parameters. However, if the body is unavailable, the instantiation produced from the declaration must be further analyzed when the body becomes available, due to possible body dependencies. In this case, the linker calls a subprogram provided by semantic analysis to check for body dependencies. This analysis involves checking only the list of body dependencies against the actual parameters in an instantiation. Checking for body dependencies of instantiations is a separable function wiht semantic analysis. Separating this function allows it to be called by tools other than the semantic analyzer, including the linker and tools designed to update the state of an Ada program library.

3.3.1.3.2.1.2 Instantiation Representation

To simplify later compiler phases, SEM creates a normalized DIANA representation of generic instantiations. This normalized representation makes explicit in the DIANA representation the renaming declarations implicit in the actual parameter associations. The expanded representation does not duplicate instantiation bodies, so that code sharing is possible. Normalization generates a normalized actual parameter list, in positional order, as well as inserting the DIANA for the implicit declarations.

Instantiation bodies are references to the corresponding generic body template if it is available. The DIANA representation of the instantiation, which does not include a body representation, contains an attribute that designates the DIANA representation of the generic template. This attribute is null if the body is unavailable.

3.3.1.3.2.1.2.1 Instantiation Code Sharing

After semantic analysis, separate instantiations of a generic declaration share the DIANA representation for the generic body template. In many cases, the machine code generated by later compiler phases for generic instantiations also can be shared. As examples, the representation in many cases can be shared for generics having no parameters, and for those having only formal objects, formal scalar types, formal access types, formal subprograms, or combinations of formal objects, formal scalar types, formal access types, and formal subprograms. Generics having formal private types often can be shared, if the size of the private type is included as a run-time parameter.

In general, SEM is unable to determine that multiple instantiations can be shared, because it is unaware of the run-time representation of data types. Thus, to simplify semantic analysis and to promote code sharing, an instantiation refers to the generic template; it does not expand the generic template replacing the formal parameters with the actual parameters. Later compiler phases determine the feasibility of code sharing, as these phases determine the feasibility of code sharing, as these phases have knowledge of run-time data and program representations. If sharing is possible, no new body is produced for the program unit. If sharing is impossible, a new body must be generated from the generic template, using the appropriate instantiation actual parameters.

3.3.1.3.3 Outputs

The SEM phase has one OUT parameter, which is a completed DIANA tree. The symbol table for the compilation unit is complete except for some storage allocation information. The SEM phase may also output error information in the event of errors (see Appendix B).

The DIANA tree that is produced is a copy of the input abstract syntax tree with additional attributes and minor modification of the tree structure (e.g., apply nodes are turned into function_call, procedure_call, entry_call, indexed, or slice nodes).

3.3.1.3.4 Special Requirements

In order to meet the speed requirements of the compiler, the SEM phase should run at 6000 statement/CPU-minute. Modifications to the passage of data between LEXSYN and SEM have already been discussed in the special requirements section of LEXSYN. In order to speed up compilation, it may be necessary to place a limit on the size of a separate compilation unit. This limit enables all of the data to be core resident. This would eliminate the need for VMM paging, except for symbol table entries and static value information. This limit would apply to the IBM 370, but not to the PE 8/32 machine. Since the memory size for the PE is smaller, placing a limit on source size in order to limit paging is probably not feasible, and the 8/32 version of the compiler will run slower than the IBM 370.

3.3.2 Middle Part

3.3.2.1 GENINST

The GENINST phase implements generic instantiation following semantic analysis. GENINST determines if instantiations can share generic implementations that have been generated for a previous instantiation of the given generic. Sharing may be impossible either because no previous implementation has been generated or because no previous implementation is suitable, due to dissimilarity of the instantiation actual parameters. If sharing is impossible, GENINST generates a new instance of the generic from the generic body template. This new instance is then available for possible later sharing with subsequent instantiations.

3.3.2.1.1 Inputs

GENINST has one in out parameter, the DIANA tree produced by SEM. Although GENINST adds information to the DIANA representation of instantiations, the original DIANA is preserved to ensure source reproducibility.

3.3.2.1.2 Processing

GENINST is responsible for creating appropriate DIANA subtrees for the expansion and code generation of instantiations or, if possible, for the sharing of previously generated instances.

When generating expansions for an instantiation, GENINST uses the generic body template to create a DIANA subtree that is like the template, but which is augmented with attributes describing the instance. For example, subprogram calls may be marked as indirect to enable the generated instantiations to call a subprogram passed as a run-time parameter. Other attributes regarding object size also may be added to the DIANA.

Whenever GENINST creates a new instantiation body, the body is added to a list associated with the generic template. This list contains the instance bodies generated for this template. Associated with the each instance body is a list describing the actual parameters that are possible to use with that instantiation. Upon later invocation, GENINST searches that list to determine if it is possible to reuse that body instead of generating a new body template instantiation. If the representation of the actual parameters matches those of an instance body generated previously, that body is reused. If no matching body is found, a new instance body is generated.

Thus, GENINST is responsible for either creating a DIANA sub-tree similar to the generic body template, but with additional constraints to be obeyed by later optimization and code generation phases, or for deciding to reuse a previous instantiation.

3.3.2.1.3 Outputs

GENINST has one in out parameter which is the DIANA tree produced by SEM.

3.3.2.1.4 Special Requirements

It may be necessary for the linker to invoke the compiler for generic instantiation in the case of separately compiled generic bodies. In this case, GENINST is unable to determine the body to be used with an instantiation, because the body may be unavailable. This determination must be made when the program is linked during program build. Thus, the linker calls GENINST when processing an instantiation having as associated body that is separately compiled.

On the average, GENINST will process 24,000 statements/minute. This average includes both source programs that perform instantiations and those that do not. In the former case, GENINST processes only DIANA generic declarations and instantiations nodes. In the latter case, a DIANA attribute indicates that no generics are instantiated and GENINST processing is bypassed.

3.3.2.2 STATINFO

STATINFO (STATic INFOrmation gathering) adds information to DIANA for three purposes:

(1) It initializes attributes for later Middle Part phases.

(2) It produces a call graph and symbol cross reference.

(3) It initializes attributes for the FLOW optimizer.

The particular processing is determined by the LIST and OPTIMIZE compiler options and by the OPTIMIZE pragmas in the unit.

3.3.2.2.1 Inputs

STATINFO is given the updated DIANA graph produced by GENINST, for a single compilation unit. The LIST and OPTIMIZE compiler options are present as attributes of the DIANA compilation_unit node.

3.3.2.2.2 Processing

STATINFO processes the tree in a single top-down traversal. The actions taken depend upon the class or type of the node. Each attribute created by STATINFO has a name beginning with the mnemonic "si_".

3.3.2.2.2.1 DEF_ID

If the LIST XREF option is given, then STATINFO creates the si_refs attribute for the DEF_ID node. This attribute's value is a set of nodes, and it is initialized here to be empty. At the completion of STATINFO, the set will contain those USED_SYMBOL nodes (in the current compilation unit) that are references to the given DEF_ID. If the LIST XREF option is given, then STATINFO also creates the si_calls attribute for a procedure_id, function_id, def_op, package_id, task_id, variable_id (for a task object), and entry_id. This attribute references the "call graph": the set of subprograms and entries that are called from either a subprogram, a package, a task, or an entry/accept. The si_calls attribute is initialized here to be empty. At the completion of STATINFO, the set will contain those USED_SYMBOL nodes both internal and external, for declared entities that are invoked in the immediate scope of the given DEF_ID.

In Ada, it is common for the same entity to have more than one declaration point and therefore multiple DEF_ID nodes. The attributes created by STATINFO have different values in general for each of these nodes. (For example, the call graph for a procedure specification is simply the set of functions invoked in the header, whereas for the procedure body, the call graph also includes entities called from the body.)

3.3.2.2.2.2 COMPILATION_UNIT

If the LIST XREF option is given, then STATINFO creates the si_external_refs attribute for the compilation_unit node. This attribute represents the cross reference to symbols in separate compilation units. The si_external_refs attribute is initialized here to be empty. The value of the attribute is a set of pairs. At the completion of STATINFO the first element in each pair will be an external DEF_ID node referenced, and the second element will be the set of all USED_SYMBOL ndoes in this compilation unit that refer to the external node.

3.3.2.2.2.3 block stm, subprogram body, package body, task body

Each of these nodes represents a block or body over which the OPTIMIZE pragma has an effect. STATINFO creates the si_opt_level attribute in the node, whose value is in the enumeration set (NONE, SPACE, TIME), initialized to the value of the OPTIMIZE compiler option. At the completion of the processing of the subtree rooted at the node, this attribute value will be that given in an OPTIMIZE pragma contained in the block or body, if any such pragma is present. Thus an explicit pragma overrides the compiler option.

Each of these nodes also represents a scope that will be annotated with a record of non-local variables and constants that are referenced, provided the LIST XREF option is given. STATINFO creates the si_global_refs attribute, initially empty. At the completion of the processing of the subtree rooted at the node, this attribute value will be the set of OBJECT_ID nodes declared outside the scope but referenced within, with an indication for any such node that is used in a "store" context.

3.3.2.2.2.4 package_decl

STATINFO sets the si_opt_level attribute in the node, initialized to the value of the OPTIMIZE compiler option. This value is then constant, since the OPTIMIZE pragma cannot appear in a package declaration.

3.3.2.2.2.5 pragma_decl

If the pragma is an OPTIMIZE pragma, then STATINFO checks that an OPTIMIZE pragma has not previously appeared in the current block or body (if it has, then this pragma is ignored and a warning message is produced) and then copies the value of the OPTIMIZE pragma parameter to the si_opt_level attribute of the enclosing block or body.

3.3.2.2.2.6 USED_SYMBOL

If the XREF LIST option is given, then STATINFO adds the referenced node to the appropriate attribute:

(1) If the corresponding DEF_ID node is in the current compilation unit, then the USED_SYMBOL node is added to the si_refs attribute for the DEF_ID.

(2) If the USED_SYMBOL node represents a called entity, then the corresponding DEF_ID node is added to the si_calls attribute for the unit containing the call.

(3) If the corresponding DEF_ID is in a separate compilation unit, then it is added to the si_external_refs attribute for the current compilation_unit node.

(4) If the corresponding DEF_ID is an OBJECT_ID declared in a scope containing the current subprogram_body, package_body, task_body, package_decl, or block_stm, then it is added to the si_global_refs attribute for this current unit, along with an indication whether the use is in a "store" context (target of assign_stm, or an actual in out or out parameter).

3.3.2.2.2.7 STM

STATINFO creates the si_labelled attribute in the STM node. This attribute value will be TRUE if and only if the STM is the target of a goto statement.

3.3.2.2.2.8 NAME_EXP

STATINFO creats the si_context attribute in the NAME_EXP node, whose value is taken from the enumeration set (VALUE_CONTEXT, ADDRESS_CONTEXT, PARAMETER_CONTEXT, FLOW_CONTEXT). FLOW_CONTEXT will be set for a node appearing as the expression in an if_stm or exit_stm or as the left operand to a SHORT_CIRCUIT_EXP. ADDRESS_CONTEXT will be set for a node appearing as the destination of an assign_stm, as the name in an indexed, selected, and slice node. PARAMETER_CONTEXT will be set for a node appearing as an actual parameter to a subprogram or entry call. VALUE_CONTEXT will be set in all other cases.

3.3.2.2.2.9 record_type

STATINFO creats the si_variant_index attribute (a Boolean), TRUE if the record type is for a variant record whose instances are to be represented using variant indices for tag checking (see 3.3.2.3.2.1.2). This attribute is used by the STORAGE phase of the compiler.

3.3.2.2.3 Outputs

STATINFO does not perform a tree transformation; rather, it adds new attributes to nodes. The following summarizes the new attributes. Each attribute value is stored in the program library.

3.3.2.2.3.1 DEF_ID

(1) si_refs; used by LISTER.

(2) si_calls (for procedure_id, function_id, def_op, package_id, task_id, variable_id for a task object, and entry_id); used by LISTER.

3.3.2.2.3.2 COMPILATION_UNIT

(1) si_external_refs; used by LISTER.

3.3.2.2.3.3 ITEM

(1) si_opt_level (for subprogram_body, package_body, task_body, package_decl); used by EXPAND, FLOW.

(2) si_global_refs (for subprogram_body, package_body, task_body, package_decl); used by LISTER.

3.3.2.2.3.4 STM

(1) si_opt_level (for block_stm); used by EXPAND, FLOW

(2) si_labeled; used by EXPAND

(3) si_global_refs (for the block_stm); used by LISTER

3.3.2.2.3.5 NAME_EXP

(1) si_context; used by EXPAND

3.3.2.2.3.6 TYPE_SPEC

(1) si_variant_index (for record_type); used by STORAGE

3.3.2.2.4 Special Requirements

STATINFO will process 24000 statements/minutes provided that the LIST NOXREF and OPTIMIZE NONE options are given, no OPTIMIZE pragmas appear in the unit, and the compiler is configured so that the tree part of DIANA is not paged.

3.3.2.3 STORAGE

STORAGE processes the DIANA for each type, subtype, object, component, subprogram signature, subprogram body, package, task, task type, aggregate, string literal, and all non-built-in function calls. This list contains entities that have storage allocated for them. STORAGE annotates the DIANA associated with each of those entities with a storage information node that records the layout and use of the entity; i.e., information bound during STORAGE.

STORAGE also adds nodes that can be accessed from the storage information nodes; these describe the layout of any storage associated with the entity and are called frame descriptors. The phase derives its name from this binding of a layout for entities.

STORAGE also adds nodes, again reachable via attributes of an entity's storage information node, which outline how to perform primitive operations on the entity at run time; these are called operation descriptors. The term "primitive operation" is used here to indicate one of a set of operations sufficient to compose all other operations on an entity.

The storage information nodes, frame descriptors, and operation descriptors provide EXPAND with a set of primitives. EXPAND can then transform the compilation unit into an expanded version. The EXPAND phase and DBUG both use the information about the compilation unit bound by STORAGE to access the entities in the program. The last user of the information bound by STORAGE is STORAGE itself, which looks up, in previous compiled compilation units the layouts of entities used in the present compilation unit.

3.3.2.3.1 Inputs

STORAGE takes as input the DIANA for the compilation unit as augmented by STATINFO. For STORAGE, STATINFO has marked entities as having features that enable particular optimizations. The only example of this at present is that record_type nodes are marked as suitable for the variant index optimization (see 3.3.2.3.2.1.2.)

STORAGE visits nodes in the DIANA of the compilation unit with few exceptions. The significant exceptions are the use_id nodes of typed objects in expressions and the function_call nodes of built in functions.

STORAGE visits type_spec nodes to bind a layout for type descriptors and prototype layouts for values of the type (subtype). Storage visits DEF_ID nodes for variables, constants and components to bind a layout for the object associated with the identifier. STORAGE visits HEADER nodes to bind a layout for the subprogram, and entry call sites. STORAGE visits aggregates, and string literals to bind the layout of the storage associated with the object built to hold the literal parts of the aggregate or the string.

STORAGE scans entities that may enclose other entities; subprogram bodies, blocks, packages, tasks, and task types. This scan allows STORAGE to lay out the enclosed entities and in turn to lay out the parent entity.

3.3.2.3.2 Processing

STORAGE scans the DIANA in a single recursive scan that is primarily in elaboration order. Since it is possible to use an entity prior to its full declaration, STORAGE departs from elaboration order to process a private type's full declaration at the point it is first introduced.

In scanning the DIANA, STORAGE performs a number of interconnected tasks; laying out entities and recording how to compute primitive operations. Since the layout of a given entity may require the layout of others (e.g., those embedded in the first entity or those used in the body of the first entity), a state vector and stack is maintained for snapshotting the state of tasks in progress when the mechanism of STORAGE recurses.

Typical of entries in this state vector are those present when laying out a record. Records are processed in two stages. The record is scanned to collect the layout of each component. The set of component layouts is maintained in the state vector. In the second stage, each component is assigned an offset in the record (this activity is known as packing). In processing a record, it is necessary to push the set of accumulated components whenever a variant part is encountered. The scan for records, collection of component layouts, followed by packing, is generalized in STORAGE and used for all entities that enclose other constructs. Thus packages, subprograms, and tasks are all scanned to accumulate a set of enclosed entities and then having laid those out individually, they are packed into a layout for the immediate enclosing declarative region.

STORAGE decorates each type with a prototype layout for values of that type. This prototype is used to guide the layout of objects of the type. This design allows STORAGE two distinct points in its processing to address the task of laying out a given typed object: first, when the type is laid out, when issues local to the type can be addressed; and second, when the object is declared, when issues local to the declarative region of the object can be addressed. An enumeration type will provide a simple example.

Enumeration types are always laid out into a minimum sized bit field. When objects of the enumeration type are declared, that minimum sized field may be enlarged to ease accessing. Thus, an enumeration value which can fit in three bits can be packed into three bits in a packed record, into a half word when a local of a subprogram, and into a byte when a component of an array.

This distinction between prototype and actual layout is used throughout STORAGE. In addition to being used between types and objects of the type, it is used on other pairs related to each other in a similar definition. Thus, generics are decorated with a prototype of the data structure that should be created each time an instantiation of the generic is created. Subprogram signatures are given a prototype that is instantiated at each call site.

In the discussion which follows we review each class of entity in the language. The discussion of array types for example includes a discussion of array objects, much as the discussion of call sites is included in the discussion of subprogram signatures.

3.3.2.3.2.1 Type Declarations

In visiting the DIANA tree associated with a type (subtype), STORAGE lays out a descriptor for the type and a prototype layout for values of the type, adopting a minimum descriptor approach to the creation of descriptors. If an entry in a descriptor can be recomputed thoughout the scope of the type (subtype), then it is not entered into a descriptor. Constant bounds for subtypes are not placed in descriptors and subtypes declared in record declarations are never given descriptors.

For each type and subtype in the compilation unit STORAGE creates a storage information node. In addition to an attribute which records the descriptor's layout, STORAGE also fills out attributes describing how to perform primitive operations on the type (subtype). Examples of such operations include computing SIZE and FIRST for a type (subtype), or how to initialize the descriptor.

When it visits the declaration of an object of the type, STORAGE annotates that object's DEF_ID node with an attribute that, again, points to a storage information node. There, STORAGE records the layout selected for the object and how to perform a set of primitive operations on the object. Examples of such operations are computing attributes such as SIZE, ADDRESS, and intializing the object.

3.3.2.3.2.1.1 Scalar Types

Scalar subtypes are all given a descriptor with entries for each bound, if that bound can not be safely and quickly recomputed during the scope of the subtype. The storage information node for a scalar subtype records how to compute those bounds. The Front End of the compiler must be able to determine some attributes of static scalar subtypes to enable it to compute static expressions. This may require certain layout choices to be made during the SEMANTICS phase. To enable this, the storage phase provides a package of routines for use by the Front End to process scalar type declarations.

3.3.2.3.2.1.1.1 Integer Types

In accordance with Ada Language rules, STORAGE selects one of two built-in types for each integer type; i.e. small_integer (a half word on the IBM 370), or integer (a word). This built-in type has a layout recorded on it (for example integer is a signed word) and that layout is given to the new type. Objects of this new integer type are aways layed out in a frame of that size, independent of packing.

3.3.2.3.2.1.1.2 Enumeration Types

All enumeration types are treated uniformly; i.e. Boolean types, character types, and simple enumeration types. Enumeration types are given a prototype layout which is a bit field as small as possible (compatible with the range of values the enumeration can take on even given a representation specification). This prototype can then be enlarged when objects of the enumeration type are created.

Some enumeration types also require the creation of a descriptor that holds a data structure to guide the mappings done in computing the attributes IMAGE, VALUE, VAL, POS, SUCC, and PRED. As STORAGE scans the representation clause's DIANA for a given enumeration type, it lays out this data structure and creates operation descriptors outlining how to compute those attributes. This need only be done for each ground type and each type that was given representation clauses.

3.3.2.3.2.1.1.3 Floating Point Types

For each floating point type, STORAGE selects one of the two built-in types provided for floating points; i.e. float (a single word floating point on the IBM 370), and long_float (a double word floating point). The smallest possible floating point is always selected. An error message is issued if the accuracy requested is too large.

The storage information nodes associated with a floating point type have operation descriptors that outline how to compute the many attributes of a floating point type; MANTISSA, SMALL, etc.

3.3.2.3.2.1.1.4 Fixed Point Types

All fixed point types are a single word long. The position of the binary point is selected to be compatible with the accuracy constraint given on the fixed point type's immediate subtype. STORAGE will issue an error if it is unable to achieve the accuracy requested.

The STORAGE information node associated with a fixed point type has operation descriptors that indicate how to compute the many attributes of a floating point type; DELTA, WIDTH, IMAGE etc.

3.3.2.3.2.1.2 Record Types

Record types have a descriptor laid out for them that records the value of subtype bounds in the record's declaration which were based on expressions that cannot be safely recomputed during the scope of the record type.

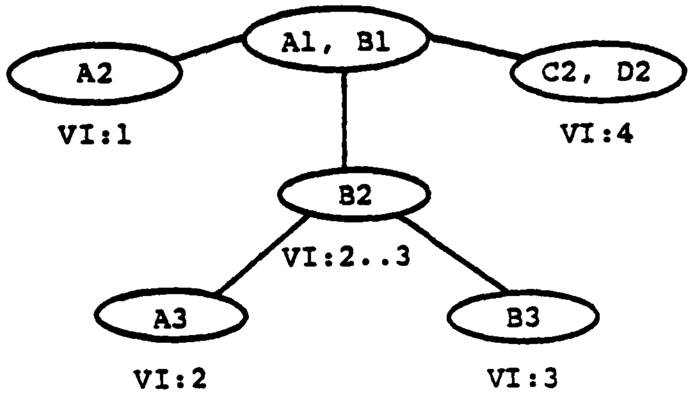
The significant processing done on a record is the scan and packing of the record's prototype layout that was outlined earlier. The simple outline given there is complicated by the presence of variants in the record.

The accessing of a record component in a variant must be prefaced by a test that this particular variant has that component present. This test can be very complex. In some cases it is advantageous to include an additional field in the record to simplify the test. This field is called a variant index and it records which branch of the variant this record contains. The variant index is then computed once when the record value is created, the test is then always a simple range check.

A variant record type may be regarded as a tree of variants; each internal node represents the sequence of component declarations preceding the variant part, and there is a subtree for each clause of the variant part. For example, consider the following declarations:

```
subtype S is INTEGER range 1..20;

type T(M,N: S :=3) is

     record

     A1: ...

     B1: ...

     case M of

        when 1 => A2: ...

        when 2..5 | 7 => B2: ...

                           case N of

                              when 2..6 => A3: ...

                              when others => B3: ...

                           end case;

        when others =>  C2: ...
                        D2: ...
     end case;

     end record;
```

This produces the tree:



The leaves of the tree are labeled from left to right with Variant Index (VI) values, and parent nodes synthesize the resulting range. When an instance of the record type is created, a specific Variant Index is computed. This index is set in the discriminant descriptor. When a field is selected, this index is compared against the range of VI values for the field (this range is present in the type descriptor). For example, the declaration:

```
    X: T;
```

results in the Variant Index value 2 (since M and N are both 3). Thus, only the selections X.B2 and X.A3 are valid; all others raise CONSTRAINT_ERROR.

The desirablity of the variant index optimization is determined during STATINFO. STATINFO leaves a flag in the record declaration marking the choice it made. To avoid adding additional fields to a record for which the user has given a representation clause, the variant index is not always generated. The varient index is also not generated if tests are already simple.

The storage for disjoint variant parts is overlaid. Discriminants are packed to be contiguous, to enable block compares against the discriminants of other record values. Dynamically sized record components are always placed on the tail of the record to ease accessing of the statically sized components. To ease accessing any dynamically sized objects, an offset pointer is placed in the record's static component area.

The operation descriptors generated for a record indicate how to compute various attributes of the record type (SIZE, for example) as well as how to access and initialise the descriptor.

Like all entities, the components are also given storage information nodes. The operation descriptors for those describe how to initialize each component and how to access them. As part of that accessing descriptor, the appropriate component present test is included.

3.3.2.3.2.1.3 <u>Array Types</u>

There are never any array type descriptors; array subtype descriptors may record bounds for the index subtypes. These index subtype descriptors are identical to scalar subtype descriptors. The storage information node for an array type records how to index components in the array.

Array components are instantiated from the prototype of their value's type in one of two ways, packed or not. Packed arrays are always laid out to minimize the amount of storage they consume, leaving only waste at word boundaries.

For nonpacked arrays, bit fields are enlarged to ease accessing; a 7-bit field is enlarged to fill a byte to allow the code to take advantage of string manipulating instructions when accessing the array.

Array objects fall into two classes, statically sized (i.e., those whose size is known when STORAGE processes them), or dynamically sized. Statically sized arrays are instantiated from the prototype without change.

Dynamically sized arrays are handled differently in records vs other situations. In records the dynamically sized objects are placed on the tail of the record. Dynamically sized objects in all other contexts are allocated on the secondary stack. A pointer to the object is then placed in the package or subprogram's static data area.

3.3.2.3.2.1.4 Access Types

Access types have storage associated with them for both a descriptor and for a collection, if they require it. The descriptor may contain fields for managing the collection. When STORAGE scans the declaration of an access type it must lay out that storage. This is done by creating an instance of a prototype maintained in an internal catalog that indicates how access types managed by a particular collection management scheme are to be laid out.

The operation descriptors for an access type describe how to perform a set of primitive operations that include: allocation, deallocation, and initializing the collection associated with the type.

When laying out the accessed values, STORAGE must ensure, as it does for array elements, that the size of the value is compatible with its alignment requirements. In addition, particularly small objects such as bit fields, may be enlarged to allow the collection management scheme to place the data it needs into unallocated values (pointers, sizes, etc.).

Finally, having laid out the access type descriptor and the accessed value, STORAGE must lay out the access value prototype. In all cases, this is a full word pointer in the IBM 370 implementation.

3.3.2.3.2.2 Nontyped Entities

The layout of entities other than typed objects is similar to that of records; i.e. the entity is scanned, a collection of embedded objects is accumulated, and they are packed into a layout for the entity. Each such entity's DIANA is decorated with nodes to record that layout and a set of primitive operations. These primitive operations include how to compute attributes such as ADDRESS, or SIZE, as well as how to initialize the data structures of the entity.

3.3.2.3.2.2.1 Packages

Each library unit package has as many as three areas of storage associated with it; a read-only area that holds constants and literals, a read-write area that holds statically sized globals, and a portion of the secondary stack of the package's enclosing task where objects whose size is not known until elaboration/run time can be allocated.

Packages embedded within a library unit package have their storage merged with that of their parent. Having scanned such a package, STORAGE skips packing the set of embedded entities and instead "hoists" them into the set of entities of the parent.

3.3.2.3.2.2.2 Subprograms

A subprogram has two kinds of data areas associated with it, globals and locals. There are two global data areas: a read-only data area and a read write data area. The locals area divides into three parts: a call site, an area for locals that are statically sized, and one for locals whose size is not known until run time.

The global data areas of a subprogram are hoisted into the global data areas of the compilation unit in the same way that embedded packages have their global data areas hoisted. If the user has specified the pragma STATIC for the subprogram then STORAGE merges the locals with the read write global data area.

3.3.2.3.2.2.2.1 Signatures and Call Sites

The call site of a subprogram is referred to as a frame header. STORAGE lays out the frame header when it scans the signature of a subprogram. This region has two components. The first is the parameter area. The second is allocated for the Run Time System which uses that area for linkage pointers and for the register save area.

Having visited the signature of a subprogram, STORAGE may then allocate storage for its call sites. These are laid out by instantiating the frame header in the same way a prototype value frame for a record type is instantiated. If the user has specified the pragma STATIC for the subprogram, then STORAGE does not instantiate a new call site but sets up the association to the static call site of the subprogram.

3.3.2.3.2.2.2.1.1 Layout of Parameters

STORAGE treats return values like OUT parameters for layout purposes. The exception to this rule involves subprograms that return variable sized return objects. These are not identical to

OUT parameters since an OUT parameter will always have an actual of known size prior to the call. Variable sized return objects are allocated on a secondary stack by the subprogram and deallocated by the caller following the return.

The layout process for parameters is only roughly similar to the layout process for a record. Like the processing of a record, the set of parameters is scanned and, for each parameter, a layout is instantiated based on the prototype value layout recorded on the storage information node of the parameter's type. Unlike the record component instantiation process, the parameter instantiation process can generate a layout very different than that given for the prototype value.

Dynamically sized objects are passed by reference. A descriptor must be created for such objects. The subprogram's body will use the descriptor to enforce constraint checking. For an unconstrained record parameter, this descriptor consists of a bit, used to indicate if the value passed was constrained or not. For an unconstrained array parameter, a descriptor of the array's bounds must be created.

Objects whose size is statically known to the subprogram are divided into two classes, small and large. Small objects, which include all scalars, access values, small records, and small arrays, are passed by value. Large objects are passed by reference. The boundary between small and large is 64 bits.

When STORAGE lays out a call on the subprogram, it must again consider each parameter. For those with descriptors being passed by reference, it determines if a descriptor already exists; if not, it must lay one out in the caller's locals. It then must record how to initialize that in all cases except the descriptor of a dynamically sized return value.

The storage information node for a parameter, in addition to describing how it is laid out in the frame header, also indicates how to initialize and finalize the parameter. For values passed by copy this includes doing the appropriate copying, for parameters that use the secondary stack, it includes doing the appropriate allocations and deallocations.

3.3.2.3.2.2.2.2 Subprogram Bodies

The storage associated with a subprogram's locals is called the subprogram frame. One part of that storage is the frame header, discussed in the previous section. The frame header is allocated and initialized by the caller. For the IBM 370, it is advantageous to keep stack frames statically sized. To achieve this, dynamically sized locals are allocated on a secondary stack. That same secondary stack is used to hold dynamically sized return values.

Once subprogram frames are statically sized, then one can have only a single stack frame allocation in each subprogram's prologue. To allow this, all statically sized data structures local to the subprogram must be laid out together and allocated once. In particular, the call sites of any subprograms called from within the subprogram being laid out must also be laid out.

STORAGE processes the body of a subprogram in a single scan that accumulates the layouts of any enclosed entities; local variables, and constants, type (subtype) descriptors, enclosed blocks, current exception handler slots, call sites, and aggregates built local to the subprogram. For each such entity, a layout is generated, by instantiating a prototype layout.

Storage is overlaid for those entities declared in disjoint blocks. All call sites are laid out in a block of storage at the tail of the subprogram frame, known as the call area. That area is the size of the maximum sized frame header of any called subprogram.

3.3.2.3.2.2.3 Aggregates

STORAGE processes aggregates to determine if storage is required for them and, if so, to lay out that storage. In cases where the aggregate is the initial value of a global it may not be necessary to allocate separate storage for the aggregate.

Storage allocates large literals to hold aggregates in one of two ways. For record aggregates, an exact copy of the value denoted by the aggregate is created. For array aggregates, a literal is created for each entry in the array aggregate that describes a region of the array.

3.3.2.3.2.2.4 Tasks

The layout of a task is similar to the layout of a subprogram. Some part of the storage laid out for the task is provided to allow the Run Time System, which is responsible for managing tasking, to maintain the data structures associated with the task.

For each tasking construct (task or task type declarations, entry declarations, accept statements, select statements, and entry calls), STORAGE allocates an instance of a data structure designed as part of the Run Time System for that task.

If the user has specified the pragma MONITOR for the task, then the task is laid out so as not to require a stack. The details of how the task is laid out in this and the normal case are described in AIE(1).KAPSE(1).

3.3.2.3.2.2.5 Generics

The GENINST phase of the compiler selects the mechanism used to implement a particular generic. In processing a generic instantiation, STORAGE need only instantiate a copy of the prototype for the table that parameterizes any shared code body for used by the instance.

3.3.2.3.3 Outputs

STORAGE creates, for every entity in the Ada compilation unit, a node that reflects the information that STORAGE bound about that entity. These nodes are then associated with existing nodes in the DIANA that declared those entities. These per-entity nodes are called storage information nodes.

A single DIANA node may generate multiple entity nodes; a type declaration generates a node for the type and one for the subtype. Some DIANA nodes may only occasionally generate a new node; an OBJECT_TYPE node, for example, only generates a new subtype information node if a new subtype is being defined.

All of the additions made by STORAGE to the DIANA tree are added to the program library record of the compilation unit. This enables other compilation units to use entities declared in this compilation unit without having to "simulate" the choices that STORAGE made in some previous compilation. The attributes of a storage information node record the layout of entities in frame descriptors and indicate how to perform primitive operations on those entities in operation descriptors.

There is extensive sharing in the output of STORAGE. All objects of a type may share the same frame descriptor. All arrays of a given type will share the same operation descriptor outlining how to index them.

3.3.2.3.4 Special Requirements

STORAGE spends the vast majority of its time in the processing of declarations. To enable STORAGE to process the compilation unit at 24000 statements/minute, we assume that the compilation unit has no more than 15% of its nodes in declarations.

3.3.2.4 EXPAND

The purpose of the EXPAND phase is to lower the semantic level of the program tree, making it more machine-oriented and less Ada-specific. This has the effect of exposing address arithmetic for subsequent flow optimization, as well as localizing to this phase many of the run-time system decisions. The output of EXPAND is the low-level BILL tree.

EXPAND transformations fall into several categories, as described below.

(1) Data references: an explicit contents node appears when a value is to be fetched from an address. Each address computation is transformed into an explicit arithmetic subtree, both for local/up-level variable references (up-level addressing is the accessing of objects declared in enclosing subprogram frames) and for array/record components. Thus, EXPAND produces a tree that reflects stack versus static data allocation and also the placement of data within call frames and static storage. Ada attributes (such as 'FIRST), are expanded into the appropriate data references.

(2) Object creation: declarations, allocations, initializations, and aggregates are mapped to lower-level constructs. Data that are implicit at the source and DIANA level, such as subtype constraints, are mapped to explicit creation constructs.

(3) Subprogram calls and returns: a closed call is transformed into a subtree that makes explicit the parameter binding choices (copy vs. reference). Stack manipulation by caller and callee is made explicit for closed calls. An inline call additionally results in the production of a block for the subprogram body.

(4) Checking: "checking" subtrees are generated for those nodes that implicitly require run-time checks (such as assignment, indexing, variant selection, and nested subtypes). Declarative information and the SUPPRESS pragmas are used to avoid the generation of unnecessary checking. More complete optimization of constraint checking is performed by the FLOW phase.

(5) Tasking: nodes corresponding to Ada tasking primitives are transformed to lower-level constructs and calls on run-time support routines.

3.3.2.4.1 Inputs

The input to EXPAND is the DIANA form as augmented by previous phases in the Middle Part. Information added to the "symbol table" nodes in DIANA determine the nature of the expansions of the DIANA program tree.

3.3.2.4.2 Processing

EXPAND performs a top-down traversal of the program tree. A summary of the transformations, keyed to Ada language constructs, is given below.

3.3.2.4.2.1 Lexical Elements

A catenation involving character strings and character type elements is folded into a single character string. The following pragmas are interpreted by EXPAND:

(1) CONTROLLED

(2) MARK_RELEASE - see AIE(1).KAPSE(1)

(3) STATIC - this pragma has the form:
pragma STATIC (subprogram_name {,subprogram_name }).

This pragma, if given, must appear in the same declarative part as the named subprograms. The effect of the pragma is that the call frame for each of the named subprograms is allocated in static storage. The pragma is a user assertion that the subprogram is non-recursive and non-reentrant. If the subprogram contains dynamic size local data, a compile-time warning is issued and the compiler reserves as much space as is maximally required for the type (this may cause STORAGE_OVERFLOW at run time). The maximum size of these dynamically sized objects is computed like the maximum size of an unconstrained record type.

(4) INLINE- see [Ada LRM, Section 6.3]. If an INLINE subprogram calls itself, then a warning is issued and the pragma is ignored. Further discussion is in 3.3.2.4.2.5 below.

(5) SUPPRESS - see [Ada LRM, Section 11.7].

(6) MONITOR - see 3.3.2.4.2.8 below.

(7) INTERFACE - see AIE(1).PIF(1).

3.3.2.4.2.2 Declarations and Types

EXPAND processes declarations both by interrogating attributes added to the symbol table nodes by STORAGE, and by generating call nodes for the appropriate size or initialization routines.

The new symbol table attributes for a declared data object give the call-frame position for the data value (if the object has dynamic size, then the position of the pointer to the value) and the call frame position (or simply the value, if static) for the subtype descriptor.

For type and subtype symbol table nodes, new attributes give the call frame position for the type and subtype descriptors if run-time descriptors are required.

. An object declaration at run time results in the reservation of space and, possibly, an initialization. If the object has static size, then its space has been allotted in the fixed part of the call frame. Otherwise, the declaration node (variable_decl or constant_decl) is mapped into nodes that call the size function for the type, reserve stack space for the required size, assign the object pointer in the pointer area, and perform initialization of the subtype descriptor and, if an initialization expression was supplied (explicitly or implicitly), the object value.

The ELABORATION_CHECK processing (that a body has been elaborated before a call on the subprogram) is implemented by EXPAND in the following fashion; checks for task body elaboration before task activation, and generic unit elaboration before instantiation, are handled analogously.

If a subprogram specification is given (that is not part of the body) and if calls on the subprogram may be evaluated or executed before the body is elaborated, then a Boolean variable ("BODY_ELABORATED") is initialized to FALSE as the run-time effect of the subprogram specification. This variable is set TRUE as the run-time effect of elaborating the body. Any call on the subprogram that may be evaluated or executed before the body is elaborated is expanded to include a test of the BODY_ELABORATED variable. If the variable is still FALSE, then PROGRAM_ERROR is raised.

If the pragma SUPPRESS (ELABORATION_CHECK) applies to the scope containing the declaration of the subprogram specification, then the BODY_ELABORATED variable is not created. If this pragma applies to a scope containing invocations of a subprogram that has a BODY_ELABORATED variable, then no check is performed at the invocations. For a subprogram invocation occurring within or after the body, no check of the BODY_ELABORATED variable is generated since none is necessary.

3.3.2.4.2.3 <u>Names and Expressions</u>

Names are expanded so that objects in non-address contexts are descendants of contents nodes, address arithmetic is explicit (both for up-level addressing and array/record components), and checking nodes are introduced. Up-level addressing is implemented by traversal of static links.

In performing address expansion for array indexing, EXPAND does some condensation based on its knowledge of the local properties of the node. For example, the indexed node for A(I,I), where A is an array (1..10, 1..10) of CHARACTER, results in the address expression (A-11) + 11 * contents(I) where I is a variable. This is done if the si_optimization_level attribute is TIME or SPACE.

Aggregates, in the general case, are implemented as temporaries stored in the stack. In special cases, EXPAND implements optimiizations recorded by STORAGE. If each component value in a record aggregate is a static expression and if the constraints on the subtypes of the components in the record type declaration are all static, then the target representation for the aggregate is stored in the literal pool. In the case of an array aggregate, if an <u>others</u> choice is present and a static expression then its value is assigned to each array component and then the individual non-others components are assigned. Except for cases such as packed bitstrings, a discrete range occurring as a choice is mapped, in general, into a run-time for loop. If an aggregate occurs as the initialization expression for a constant or a variable, then no separate temporary is reserved for the aggregate.

When the value of an expression must be used more than once in an expansion (for example, an index expression for a packed array is required to compute both the byte and bit offsets), the first use is replaced by a "forced_cse_create" node, and subsequent references become "forced_cse_use" nodes. The acronym "cse" denotes "common sub-expression".

3.3.2.4.2.4 <u>Statements</u>

The principal expansion occurs for the assignment statement. Constraint checking, when necessary, is made explicit, and the assign node is replaced by a store node whose descendants give the location of the target and the value of the source. In general, the location of the target requires two pieces of information: (1) the starting position (byte address and bit offset); and (2) the size of the target (in bytes or bits.) If the si_optimization_level attribute is TIME or SPACE, an array assignment whose source is a catenation is optimized, where possible, to avoid generation of an extra temporary.

When a block resulting from an inline expansion is created by EXPAND, the local variables and temporaries required by the block are reserved in the enclosing frame.

In preparation for FLOW optimizations, a loop statement with an iteration clause is transformed into a test of the iteration condition followed by a loop with an exit test of the condition at the bottom of the loop. For example,

```
while cond loop stm-seq end loop;

becomes

if cond then
   loop
   stm-seq
   exit when not cond;
   end loop;
end if;
```

A for loop is treated analogously. This tranformation is useful because the test of the condition before loop entry can frequently be optimized away, and because an explicit site is created (the point before the loop) for the movement of loop invariant computations.

3.3.2.4.2.5 Subprograms

As a result of processing a subprogram declaration, EXPAND produces BILL trees for default expressions.

The processing of a subprogram body yields the size of the call frame for the subprogram. This size is stored as an attribute of the proc_id or function_id node. It may be larger than the size predicted by STORAGE, since EXPAND may create temporaries (e.g., for catenation results). The static nesting level of the subprogram is likewise stored as an attribute.

EXPAND implements the "copy" vs "reference" binding mechanism chosen by STORAGE. When a subprogram is passed (as a result of generic optimization) its address, static link, and stack frame fixed-part size are copied. If a formal parameter is implemented by reference, then an extra contents node occurs at each reference to the formal parameter in the body.

A call on a closed subprogram is expanded so that the caller reserves stack space for the called subprogram's frame header. Next, the actual parameters (or their addresses) are assigned to formal parameter nodes, and constraint checking occurs. (Thus any exception raised will be handled by the caller, not the callee.) In the event of a copy-out or copy-in-out parameter, the calculated address is made into a forced_cse_create, so that it is the correct target after the procedure returns. The EXPAND phase will pass parameters in the call frame.

A call on an inline subprogram is expanded into a block (value-returning if a function), preserving the semantics of the parameter binding as well as the semantics of up-level references within the subprogram. Formal parameters and local variables for the subprogram become local variables (with respect to lifetime but not name visibility) of the enclosing scope. The representation used for the inline body in the expansion is the tree produced by EXPAND. If a call on an inline subprogram is compiled before the body has appeared (e.g., the subprogram specification may occur in a library package) then a closed call is compiled and a warning message is issued.

If a pragma specifies a subprogram as static, then the call frame for the subprogram is reserved in static storage. However, any object initializations within the subprogram are carried out at run time at each invocation.

3.3.2.4.2.6 Packages

Storage for package data is referenced from the stack frame for the enclosing unit. This gives a consistent approach to packages, even when they occur as library units or subunits.

3.3.2.4.2.7 Visibility Rules

A renaming declaration is transformed so that the necessary elaboration and constraint checking occur. When an object (or component) is renamed, a store node is generated to assign the address of the renamed entity.

3.3.2.4.2.8 Tasks

Tasking constructs are expanded into calls of run-time system routines. Rendezvous are executed in the stack of the caller. If the pragma MONITOR(T) is specified for a task or task type T, code outside the accept statement in the body of T is also executed in the stack of the caller. See also AIE(1).KAPSE(1).

3.3.2.4.2.9 Exceptions

The raise node is transformed in general into a call on the run-time Raise routine, and handlers are expanded into a sequence of statements. See also AIE(1).KAPSE(1).

3.3.2.4.3 Outputs

The output of EXPAND is a new, low-level program tree (BILL). EXPAND does not modify its input tree. The structure of BILL is defined in AIE(1).COMP(1).BILL(1).

3.3.2.4.4 Special Requirements

EXPAND will process 8000 statements/minute, assuming that the compiler is configured such that the tree part of DIANA is not paged. EXPAND is produced in Ada by the Bonsai processor that reads a pattern-matching notation as described in AIE(1).MGS(1).

3.3.2.5 UTILITIES

The UTILITIES package contains a Middle Park Driver called from the compiler driver, and common routines required by the Middle Part phases. These will be defined when the lower level structure of the phases is established.

3.3.3 Back End

3.3.3.1 FLOW

The purpose of the FLOW phase is to perform target machine independent optimization. The name FLOW is used to denote the control and data flow analysis that is performed in conjunction with optimization. Both inter-procedural (flow due to subprogram calls) and intra-procedural (flow within subprogram) data flow analysis are performed. The results of flow analysis are used to perform the following optimizations:

(1) Constant propagation
(2) Redundant constraint check elimination
(3) Constant folding
(4) Elimination of unreachable code
(5) Movement of loop invariant code out of loops
(6) Redundant computation elimination
(7) Reducing multiplications within loops to additions
(8) Algebraic simplification of expressions and statements

Beside these optimizations, FLOW performs three computations needed by later compiler phases. FLOW computes the set of possible addressing modes (i.e., where the datum may be stored and how it might be addressed). It also labels the possible branches that boolean operators may take for short circuit evaluation of the 'and then' and 'or else' operators. FLOW marks redundant expressions, loop parameters, formal parameters, created objects from strength reduction, and expressions moved out of loops so that the later phase, VCODE, can assign them a temporary name and TNBIND can allocate them to a register through the loop. See the phases VCODE and TNBIND for a discussion of temporary names.

FLOW considers each unit within the compilation unit to be like a subprogram. Packages contained in other units have been eliminated by EXPAND. Since the effect of elaborating library packages is the same as a call to initialize their data, such packages may be treated as subprograms. Tasks, with their corresponding entry names, are considered to be one unit which is handled much like a subprogram. Hence the rest of this discussion will deal with subprograms with the realization that packages and tasks are handled similarly.

The FLOW phase design is based on optimizations that are allowed by the Ada semantics. The optimizations are constrained by Ada Language rules (Section 11.6 of the LRM) and by the effects of exceptions. Briefly, the rules allow the elimination of redundant expressions and the motion of expressions from loops; however they severely limit the motion of assignment statements or any other storage-modifying statement whenever an exception might occur. Since exceptions may occur in most Ada contexts, we have chosen not to implement optimizations that can affect the order of assignments.

3.3.3.1.1 Inputs

The major input to FLOW is the BILL intermediate tree generated by EXPAND. This tree consists of two parts. The first part is the representation of the compilation unit. It is also referred to as a BILL tree. The second part is the symbol table encoding all objects that have assigned storage and their relationships to other objects. This table is referred to as the BILLET (or BILL Environment Table).

The FLOW phase also receives (as an attribute of the BILLET entry for the compilation unit) the DBUG compiler options as input. The optimization level, as determined by pragmas and the OPTIMIZE compiler option, appears as an attribute in BILL blocks and bodies.

3.3.3.1.2 Processing

The FLOW phase is organized as three passes over the BILL tree. Each pass performs the same optimizations, making certain assumptions concerning loops and subprogram calls. This is necessary since the optimization techniques used need to know both the effects of loops before processing loops and the effects of subprograms which may not be available until the completion of the first pass over the entire BILL tree for the compilation unit.

The first pass over the BILL tree performs all functions assuming that any loop modifies all objects described in the BILLET. During this pass, it determines which objects may actually be modified in each loop. When a subprogram call is encountered, two assumptions are possible. If the subprogram being called has already been completely analyzed during this pass (and all of its effects on other subprograms and data are known), then this computed information is used to describe the effects of this call on local data flow. If all of the effects of the called subprogram are not known, the call statement is added to a call graph describing subprogram invocation for the compilation unit. The call is assumed to modify all objects that are visible to both the called and calling subprogram.

At the completion of the first pass through the entire compilation unit, FLOW has created a call graph indicating all calls to subprograms whose entire effects were unknown at the point of call. This includes recursive subprograms and subprograms whose order of declaration prevented knowledge of effects. During the first pass, FLOW also has gathered, for each subprogram, the set of objects (BILLET entries) that were directly modified by the subprogram. By iterating through the call graph until stabilization, the effects of a called subprogram will be included in the local effects of the caller. Thus, at the completion of the first pass of this computation, the effects of all subprogram invocations will be known.

The first pass of FLOW is performed on each subprogram in the compilation unit in sequence. Passes two and three (and the rest of the Back End of the compiler) are applied to each subprogram before processing the next subprogram in the compilation unit.

The second pass through the tree is performed if there was either a loop statement or a subprogram call with unknown effects in the subprogram. Thus, the pass may be performed for some of the subprograms in a compilation unit and not for others. During this pass, the correct set of objects possibly modified by a loop statement is used to indicate the effects of the statement and the correct effects of all subprogram calls are used.

The third pass will be performed if the second pass moves an expression out of a loop, generates temporary objects for reducing multiplication to additions, or improves its knowledge of the value range of objects.

The above discussion describes the assumptions which are made on each pass. The rest of the processing description will describe how each pass over the tree is performed.

Each pass is an execution order tree walk of the BILL tree. During this tree walk, the modification sets of each statement and expression are computed. This information is recorded in the BILL for loop statements and the entire body of a subprogram; however, it is used during the tree walk to determine which expressions and assignments affect the values of other expressions and assignments. The techniques for recording this information will now be described together with the optimizations performed.

These descriptions are written as if each optimization will be applied, whenever applicable. This is not the case. Before any optimization is applied, the modified tree and the original tree are compared by a machine-dependent payoff function. This function gives an estimate of the benefit that will occur from the transformation. If the benefit is large enough, the optimization will be performed. This can give only approximate results since performing an optimization can have both positive and negative results. For example, eliminating a redundant expression can decrease the number of instructions needed if there are sufficient registers in the target machine to hold the result. However, if the registers are exhausted, then such an elimination might be a pessimization.

(a) Value Numbering. Constant propagation, redundant expression elimination, and the first part of constraint check elimination are performed using value numbering. Value numbering consists of assigning a key to each expression (typically a number hence value numbering). This key only indicates the value of the expression; it is not the value of the expression. Two expressions with the same value number must have the same value. The value number of each expression and object is stored in a table called the available expression table. The index into this table is either the name of the object or the operator and the value numbers of the operands of an object. At each expression node in the BILL tree, the expression is found in the available expression table. If it is already there, then the expression is redundant. If it is not there, the expression is entered. An indicator is kept to indicate whether this expression or object in the available expression table is constant. During the expression lookup, if the expression has constant value the tree is replaced by the constant. During value numbering, various relational expressions involved in checking constraints may be eliminated, thus eliminating the constraint check.

The available expression table is maintained with auxiliary data structures that indicate the effects of entering compound statements, performing a split or fork in the control flow, or performing a join of two control flows. This information is maintained assuming structured control statements in the subprogram. The only effect of the unstructured 'goto' statement is on labelled statements which are the destination of a 'goto' statement. At such a statement, we assume that every object that can be modified by this subprogram has been modified directly before this statement. The modification information is used to keep the available expression table accurate. When an assignment statement, subprogram call, or any other statement which can modify an object is processed, the value number for each object modified must be changed to indicate that all expressions involving this object as an operand are invalid.

(b) Constant Folding. Whenever the value of an expression can be determined at compile-time using value numbering the expression is replaced by its value. Complex statements involving this expression will also be simplified. Thus an 'if' statement with a conditional expression having value true will be replaced by its 'then' part.

(c) Code Motion. Any expression whose operands are not modified in a loop and which is computed each time through a loop is called a loop invariant expression. Such expressions can be precomputed before the loop. To do this, each expression in the available expression table is assigned a level number. This level number is the maximum level number of each of its operands. The level number of an object is the loop nesting level of the current assignment to that object. The level number of an expression indicates the loops out of which that expression can be moved. If this expresion is on every path from the beginning of the loop, the expression will be moved to a point prior to the loop body. The EXPAND phase has recast loop statements so that the body of each loop statements in the BILL will be executed at least once. Hence, this code motion will introduce no extraneous exceptions.

When FLOW decides to move an expression out of a loop, a copy of the expression is inserted before the loop and is marked to signal VCODE that a TN should be assigned to it for later register allocation. The next pass through FLOW (the second or third pass) will eliminate all uses of the expression inside the loop by redundant expression elimination.

(d) Algebraic Simplifications. During the tree walk, each expression will be simplified by applying algebraic identities. This will be done only if none of the operands can have side effects, such as function calls. Such optimizations as replacing A*0 by 1 are included here.

(e) Strength Reduction. During the processing of a loop body, FLOW maintains a list of multiplication expressions where one of the operands is an object and the other is a loop invariant expression (see code motion section) and the multiplication occurs on every path from the beginning of the loop body. If the object involved is modified once in the loop by simple incrementing or decrementing by a loop invariant expression, the compiler will introduce a new object to maintain the value of the multiplication. This object is initialized before the loop and is incremented following the modification of the original object. All occurrences of the multiplication are replaced by this new object (hence, the multiplication has been replaced by a repeated addition). If all references to the original object have been eliminated, the object will not be initialized before the loop. The loop exit test may also be eliminated if the newly created object can be used in place of the original one.

(f) Constraint Check Elimination. FLOW maintains a table recording the value range of objects in the BILLET and indicating whether an access variable is not null. This table is also maintained across forks and joins in the control flow. Here the effects of two separate paths at a join are combined rather than discarding the effects of both as in the available expression table. For range information, the range at a join is the union of the ranges on each path. For access variables, the non-null status of an object is true if it is non-null on both paths reaching the join.

At a fork in control flow, the range information for each path is modified to indicate the reason for the fork. Range information for a loop statement is maintained with the modification information. During the first pass, worst case estimates are used for the value range of objects. During subsequent passes, the computed range information at the end of the loop body (from the preceding pass) is merged with the value range information entering the loop. This gives range information at the beginning of the loop body.

If the range information for an object indicates that a constraint is redundant, the constraint check is eliminated. In any case the range information from the constraint check can be used to improve the range information for the constrained object.

(g) Addressing Modes. FLOW estimates the ways a particular value can be addressed during code generation. This is done using a hash table generated by the code-generation table builder. This table is searched using the operator and the sets of addressing modes of the operands as a key. The result of the search is the set of addressing modes for the tree node. There are few possible addressing modes sets, so the sets will be identified by a small integer indicating which set rather than a bit string.

(h) Control Flow. During the first tree walk, the control flow of the 'and then', 'or else', and 'not' operators is identified for later code generation. This is done during the top-down tree walk by recognizing the contexts in which such operators occur and assigning indicators for machine labels as attributes of the expressions and statements involved.

(i) Expression Reordering. During each of the three passes over the tree, the register complexity of each expression is estimated. This estimate is made using a simple model of code generation where each value is computed in a register, while objects may be in registers or memory. Each expression is reordered, subject to Ada semantics, to minimize the register complexity measurement. Typically, complex operands will be computed before simpler operands. This computation will be parameterized by a hand-built table indicating the forms of machine instructions available on this machine. This table may later be generated by the table builder (AIE(1).MGS(1)).

(j) Effects of Options. The compiler options modify or inhibit certain of the optimizations as described in the following table.

Optimization Level	
TIME	Perform all optimizations and passes
SPACE	Perform all passes; however, eliminate strength reduction and make the payoff functions more critical of space usage.
NONE	Perform only one FLOW pass.

DEBUG Option	
BREAK	No effect on optimization.
ALTER	At statement boundaries,destory available expression table and eliminate strength reduction and code motion.
OFF	No effect on optimization.

3.3.3.1.3 Outputs

The output of FLOW is an updated version of the input tree. The tree has been modified in several respects. The tree may have been reformed to replace expressions or statements by simpler trees such as constant expressions being replaced by the resulting constant. Redundant expressions have been identified and marked by filling the redundant expression attribute of each address and value tree node with an indicator of the tree node which originally computed this value. The tree has been modified by moving expressions out of loop bodies. The temporary name fields for redundant or moved expressions has been assigned a value. The flow effects of 'and then' or 'or else' operators are recorded by identifying their destinations on true and false values.

3.3.3.1.4 Special Requirements

The FLOW analyzer has a goal of processing 6000 statements per minutes. FLOW has been designed to be a fast and effective optimizer. The three pass structure with the possible elimination of one or two passes was created to enhance the speed of optimization without effect on the quality of optimization. For most simple subprograms, FLOW will require only one pass to perform optimization. If the compilation unit contains loops, two passes are necessary. The third pass is required only when loops and complex interactions between subprograms occur.

During a particular tree walk, there is a fixed amount of processing per node in the tree. Thus, a tree walk occurs in time linear in the size of the tree. For each subprogram within a compilation unit, the entire BILL tree will be resident in core during each tree walk. Thus the overhead of the VMM mechanism will be minimized.

3.3.3.2 VCODE

The function of the VCODE (Virtual Code) phase is to determine register usage. VCODE operates on one BILL unit at a time. A BILL unit corresponds roughly to an Ada subprogram, package, task, or entry/accept body. (Some Ada subprograms and packages are not BILL units since they are expanded in line or hoisted to an enclosing scope.)

VCODE simulates the action of the code generator. Instead of generating code it creates a map of register needs and register usage. This structure will be used later by TNBIND to allocate registers.

VCODE chooses a particular code sequence based on the data type and range of operands. Note that VCODE does not choose exactly which instructions to use; that will depend on the choice of registers.

3.3.3.2.1 Inputs

The input to VCODE is the BILL tree produced by FLOW. The attributes which FLOW has initialized for access modes and for labels of expressions in flow context are essential for VCODE to execute properly. Other attributes initialized by FLOW such as CSE information and evaluation order are not essential in that VCODE will still execute properly - only those CSEs created by EXPAND will exist and a default evaluation order of tree order will be assumed.

The BILLET is not referenced at all with the single exception being the BILLET entry for the BILL unit which is the root input tree.

3.3.3.2.2 Processing

VCODE traverses the program tree in top-down reverse execution order. For each operation in the tree, it determines the register needs. This is done using the same algorithm as in the CODEGEN phase; the only difference is that instead of generating code, register information is generated.

The algorithm is an efficient variant on the Maximal Munch Method of Cattell. It consists of three modules: Select, Match, and Instantiate.

a) Select. This module selects an ordered set of templates to consider and then supplies them one at a time. The templates are reasonable possibilities in that they have the correct highest order operator. These templates are similar to those used by CODEGEN, the only difference being that the code to produce is removed and any templates which differ only in their register needs are combined. These VCODE templates are generated from the CODEGEN templates by the table builder program [AIE(1).MGS(1)].

b) Match. The templates are ordered by decreasing size. This has the effect of placing more efficient special-case templates first. Because of this ordering, the first template which matches is the one which "munches" the largest portion of the program tree, and it is chosen as the optimal template.

A pattern is considered to match a program tree if three conditions hold. These conditions are: (1) the operators of the pattern and the program tree are the same; (2) the access modes of the pattern leaves have a common intersection with the access modes in the corresponding positions of the program tree; and (3) any special restrictions on a given pattern are met. These restrictions are such things as: must not be zero, must be a power of 2, or must not be a common sub-expression.

After one successful match and instantiate, pattern matching normally continues at the leaves of the matched subtree. If, however, the subtree leaf was matched by an access mode containing implicit hardware arithmetic, then matching continues at the leaves of this access mode.

The access modes (also called address modes) of the IBM 370 include register (normal and floating point) access, immediate literal access, and memory access either with no indexing, single (base register) or double indexing (base register plus index register) with or without a displacement. Separate access modes are also identified based on the size of the object accessed, and there are additional modes for the contents (r-value) and address (l_value). Some of these modes, although not used directly to access the operand of an instruction, are useful to describe the components of other access modes. Each access mode determines the type storage it accesses and the tree of address arithemetic it represents. Altogether approximately 70 access modes are identified for the IBM 370.

c) Instantiate. When Match determines the optimal template, Instantiate creates TNs if registers are required. Actually a TN is created whenever some form of storage or storage class is required. Storage classes include various alignments of memory and different types or classifications of registers (single register, register pair, odd register, etc). Although an immediate literal may be considered storage, these do not cause the creation of TNs. The storage classes needed are determined by the access mode(s) which the pattern matches. If more than one access mode matches, the one which represents the largest tree is chosen. In the case where all matching modes represent the same size tree the TN is set to represent a set of possible storage classes.

An EvalTn (for evalution tn) is created for the root of the match if the result of the operation must be evaluated in a register, and a SaveTn is created for any leaves if the operands must be in registers. In some cases the EvalTn of the root can be reused as the SaveTn of one of the leaves; on the IBM 370 an AR R1,R2 instruction uses the same register, R1, as both an operand and the result. If an interior node of an expression has a distinct SaveTn (when it is used as a leaf) and EvalTn (when it is considered a root) then these TNs are preferenced. Each BILL node and Tn is marked as to which pattern it matched.

The efficient variant of Maximal Munch includes the use of so-called "failure links" to perform fast pattern matching. This is a way of combining identical parts of patterns to avoid unnecessary re-matching of similar patterns. For example, if we have determined by attempting to match a pattern that the first operand subtree is not a particular operator, then we can skip over all other patterns with the same requirements for the first operand subtree.

Register analysis is performed in a single reverse execution order walk. During the walk, patterns are applied to the expression trees to determine what code might be generated. Longer patterns are applied first, to determine special cases before general cases. For example, 2**(I+K) can be performed in a single shift operation when I is a non-negative expression and K is a non-negative constant. On the IBM 370, I is evaluated in the base register, and I+K must be less than 128. This pattern is applied before the more general pattern, 2**I, or the yet more general pattern, J**I. Once a particular pattern matches, the registers needed by that code template are recorded for later use by TNBIND. This information is recorded by creating a structure of temporary names (TNs). The resulting algorithm is a target machine independent interpreter driven by target-dependent tables.

The information gathered by VCODE allows TNBIND to know which nodes require which registers and allows CODEGEN to select actual machine instructions.

3.3.3.2.3 Outputs

The output from VCODE is an updated version of the input tree, including attributes for temporary names and an attribute to indicate which VCODE template was matched. The data structure of temporary names includes which storage class is required (for example single register of register pair), which other TNs it is preferenced to, and which template matches it was involved in. In addition, a sequential list of all TNs created is produced. This list is in reverse order of creation which is usually execution order, the only exception being that global TNs created first by FLOW are at the end of the list. All these data structures are used only by TNBIND and CODEGEN and therefore can be in a temporary, non-VMM area. The Mark-Release heap will be used; at the beginning of the separate processing for each BILL unit (during FLOW) the heap will be marked, and at the end after FINAL will be released.

Summary of attributes:

for BILL nodes:

EvalTn:	The temporary name describing where this node is evaluated.
SaveTN:	The temporary name describing where this node is saved.
RuleMatched:	A key identifying which VCODE template matched the root of a subtree. This may indicate that an access mode tree representing hardware address arithmetic matched.

for Temporary Names:

PreferenceLinks: What other TNs this is related to.
RuleMap: The key identifying which VCODE template caused the creation of this TN and the position (Eval or Save TN).

global (for the BILL Unit):

TNs: A sequential list of all TNs in this unit.

3.3.3.2.4 Special Requirements

VCODE will process 24,000 statements/minute. The design decisions not to have TNs nor pattern templates be paged VMM objects and not to reference the VMM BILLET were made primarily to improve the speed. The fast pattern matching using failure links also speeds processing over the traditional Maximal Munch Method algorithms.

3.3.3.3 TNBIND

The purpose of TNBIND is to choose where each expression and object will be computed, thereby eliminating nonproductive data moves. The name TNBIND is a contraction of Temporary Name Binding. Here, TN will refer not only to temporaries, but also to loop parameters, formals, expressions moved from loops, and redundant expressions.

Temporary names (TNs) are created in VCODE and FLOW. In FLOW they are assigned to redundant expressions, loop parameters, and expressions moved from loops. In VCODE they are assigned to the temporary expressions computed during the generation of code. As VCODE simulates the generation of code it records all uses of TNs and how they are used.

TNBIND will take this information collected by VCODE and use it to assign TNs to registers or to the spill area in the unit's local storage. TNBIND will attempt to minimize the number of registers in use while also minimizing the occurrence of load and store instructions. To do this it will attempt to assign more than one TN to the same register. This can only be done when one TN is no longer needed at the time the other TN is computed. Furthermore VCODE recorded a preference relation indicating that it would be helpful if two particular TNs did share storage. TNBIND attempts to accomodate this request when it is feasible. Such preferences occur between one of the operands of an operator and the result. If the preference is honored by TNBIND a register-to-register move may be eliminated.

The TNs are allocated in order of importance without backtracking. VCODE has provided all information for determining the costs of allocating a TN to each form of storage. The TNs are ranked by the differential cost of storing them in registers versus the spill area in memory. Those TNs which most adversely affect the number of load instructions will be allocated first. It will be assigned to one of the registers or storage which minimizes the number of loads. This algorithm is a variant of the PQCC register allocation method created at Carnegie Mellon University by Bruce Leverett.

3.3.3.3.1 Inputs

The input to TNBIND is the BILL representation of a unit, the associated TNs created by VCODE and FLOW, a preference relation on the TNs indicating which should share storage, and a list of all code template matches occuring during VCODE and the place of each TN in the relevant matches. The DBUG option is also a parameter to TNBIND, obtained as an attribute on the BILLET node for the compilation unit.

3.3.3.3.2 Processing

TNBIND consists of two parts. First the compiler must determine which TNs conflict with one another. This is done by the live/dead analysis algorithm. Then the compiler must allocate the TNs first to registers and then to the spill area in local storage.

(a) Live/Dead Analysis. TNBIND computes live/dead information using an iterative tree walk. This is a reverse execution order tree walk which determines that a TN is live when it sees a use of the TN and marks it dead at the point of a computation. This information is stored as a bit vector, however these bit vectors need not be attached to the BILL tree except in the case of loops and labelled statements. The live/dead algorithm creates the conflict graph, a relation on the set of TNs indicating which TNs are simultaneously live. This is done at the computation of a TN. At its computation each TN that is live at that point is in conflict with the TN just computed. The tree walk proceeds, maintaining the set of TNs live at the current point. At forks in control paths the tree walking procedure takes the union of the bit strings computed on each path. At the join of a control path the tree walking procedure saves the current set of live TNs then walks each branch separately restoring the live/dead information as it starts each branch.

For loop statements the TNs live on entry to a loop body are those that are live on exit from the loop body together with those live on exit from the loop. A loop statement will require at least two walks of the tree. The set of TNs live on entry to a loop body is saved between passes as an attribute of the loop statement in the BILL tree. Initially all TNs are assumed to be dead on entry to the loop body.

'Goto' statements must be handled separately. At each labelled statement is stored the set of TNs live on entry to the labelled statement. Initially this is the empty set and on each pass it is updated from the computed set of live TNs. The TNs live on entry to the goto statement are those live on entry to the target of the goto statement.

If there are no loops or goto statements the live/dead algorithm will give correct results in one pass. If there are loop statements two passes are necessary. When a goto statement is present the algorithm will iterate until stable results occur. Also note that contrary to the modification set calculation in FLOW there are no interprocedural or aliasing effects since registers for each unit are allocated separately and saved on entry and exit from any subprogram call.

If the DBUG option ALTER is specified each TN is declared dead at a statement boundary.

(b) Allocation. Allocation of TNs occurs according to rank. Each TN is ranked according to the costs of allocating it to its preferred storage area as opposed to its second choice. The TNs are allocated in rank order with the rank being modified when a storage area is filled.

When the decision to rank a particular TN is made each storage location in the available storage area is inspected and the costs of allocating to this location are computed. The cost includes the cost of not being able to allocate a later TN to an area it would best be suited for. The TN is allocated the storage location which minimizes the cost. This location is recorded as an attribute in the TN.

When a TN is allocated all TNs with a preference relation to this TN are inspected and their rank modified to improve their chances of being allocated next. In that case the allocation of the preferenced TN will be to the same storage location since the preference relation is one of the costs used in determining which storage location to place a TN in.

If a TN cannot be allocated to the primary area of storage it needs(such as a register) it is allocated to the spill area reserved in the local storage of the unit. This spill area is allocated in the same manner that registers are allocated so multiple TNs will share the same storage location and preference information will be used as with registers to eliminate loads, stores and memory to memory moves.

This algorithm gives an approximate minimization of register costs. It is not truly minimal since register allocation is a theoretically complex issue. It works particularly well with the TNs created in VCODE since they have short lifetimes, conflict with few TNs, and do not have complex usage patterns.

3.3.3.3.3 Outputs

The output of TNBIND is the table of TNs annotated with the storage information required for code generation. This information is the physical register the TN will reside in or the location in the spill area for the TN.

3.3.3.3.4 Special Requirements

It is the goal that TNBIND will process 12000 statements/minute. To accomplish this the live/dead analysis algorithm was optimized to identify special cases. If there are no loop statements and goto statements live dead analysis can be done in one pass. If there are loop statements two passes are necessary. When a goto statement occurs the compiler will iterate until the live/dead information stabilizes. The allocation algorithm is linear in the number of TNs.

The major portion of the TNs are the temporary names created by VCODE. Typically these names have a creation point and one use. For these TNs the live/dead analysis and allocation will be particularly fast.

3.3.3.4 CODEGEN

The purpose of CODEGEN is to generate target-machine instructions. Like VCODE, TNBIND, and FINAL, CODEGEN operates on one BILL unit at a time. The job of CODEGEN is relatively straightforward due to the prior decisions that are encoded in the post-TNBIND BILL tree, as enumerated below:

(1) a completly optimized intermediate form - all optimizations not directly related to specific code sequences have been performed;

(2) the execution order has been completely determined to put minimum demand on scarce resources;

(3) the point of creation, each use, and the final use of each common subexpression has been determined;

(4) context information (e.g., needed to make a flow decision) is readily available;

(5) the access mode to be used (e.g., based off the frame pointer) has been determined for each operand;

(6) the register and storage allocation problem has been resolved; the code-generator knows exactly where its operands are and where its result must go.

The output of CODEGEN is pseudo-target code. It is pseudo in that addresses of jumps and short literals have not yet been resolved, and the instruction stream is represented as a doubly-linked list rather than as an object module. It is target code in that there is a one-one correspondence between most of the nodes in the linked list and actual target machine instructions.

CODEGEN is a table-driven algorithm. It performs a reverse execution order walk. At each step it chooses the template which matches the largest BILL tree and instantiates it.

3.3.3.4.1 Inputs

The inputs to CODEGEN are the BILL representation for a unit, including the TN attributes initialized by TNBIND, and the DBUG option (appearing as an attribute of the BILLET node for the compilation unit) that was input to the compiler DRIVER. The attribute set by VCODE indicating its pattern match will speed CODEGEN's similar match. The TN attributes set by TNBIND indicating which registers are assigned will determine CODEGEN's choice of template and code produced. The entire BILL tree is processed. However, like VCODE, the only BILLET which is referenced is that of the BILL Unit.

3.3.3.4.2 Processing

CODEGEN uses the same maximal munch algorithm as VCODE. This is a reverse execution order tree walk and consists of three modules: Select, Match, and Instantiate.

The maximal munch algorithm takes as input the BILL tree (annotated with register usage) and a prebuilt set of code templates. First the algorithm selects the largest template that matches the root of the tree and tree nodes near the root. When the match is found, the instantiate module is called to generate code. This template match divided the tree into pieces: the matched piece, and several unmatched subtrees. The maximal munch algorithm is applied to each of the unmatched pieces in reverse execution order until there are no unmatched subtrees. The resulting code is generated in reverse order.

(a) Select. This module selects an ordered set of code templates to consider and then supplies them one at a time. The templates are reasonable possibilities in that they have the correct highest order operator. The templates and their associated costs will for the most part be generated manually. The table builder program [AIE(1).MGS(1)] will perform some processing, including calculating time and space costs based on instruction time and space, sorting the templates by size and highest order BILL operator, generating failure links for fast pattern matching, and combining CODEGEN templates to produce VCODE templates.

(b) Match. The templates are ordered by decreasing size. This has the effect of placing more efficient special-case templates first. Because of this the first template which matches is the one which "munches" the largest portion of the program tree, and it is chosen as the optimal template.

(c) Instantiate. When Match determines the optimal template, Instantiate issues the associated code. While this code is being issued CODEGEN tracks the register usage as specified by TNBIND so it can allocate extra registers when needed by a template. If DBUG (ALTER) is specified CODEGEN notes that no registers have known values at the beginning of a statement.

3.3.3.4.3 Outputs

The output of CODEGEN is a doubly-linked list of pseudo target code. This is only used by the FINAL phase and, therefore, can be in a temporary, non-VMM area. In addition, a list of all jump instructions is generated for use by FINAL.

The representation of an external symbol is based on its associated DIANA pointer. This DIANA pointer is found in the BILLET for the entity. The representation consists of a combination of the DIANA pointer plus a small number n which indicates which function of this entity is being referenced. For example, the external symbol for accessing the size function of a record type is the DIANA pointer for the record type node plus some distinguishing value n. The external symbol for a routine which allocates objects of the same record type would have the same DIANA pointer component but a different value for n.

In CODEGEN, while generating code for the BILL unit, additional information is included immediately preceding the actual instructions. This information is used by the Run Time System and includes a fixed size static data area and a variably sized exception handler map. The information in this static data area includes the size of the stack frame, the size of the frame header (standard information before local variables whose size depends on such things as number of parameters, existence of dependent tasks, etc.), and the address of the exception handler map. The two sizes are available from the BILLET for the unit, and the map address is originally (in CODEGEN) set to a pseudo-code label node which is later (during FINAL) resolved into an address. The handler map is variably sized depending on the number of exception handlers in the unit. It is a table of ranges of code addresses together with the address of either the appropriate handler or the runtime system re-raise routine. Normally with an exception handler this contains three ranges one each for declarations, statements, and exception handlers. In the case of no exception handlers this is the entire range of the unit's code with the address of the runtime re-raise routine. The placement of the handler map in the object module is arbitrary since it is pointed to by a fixed location.

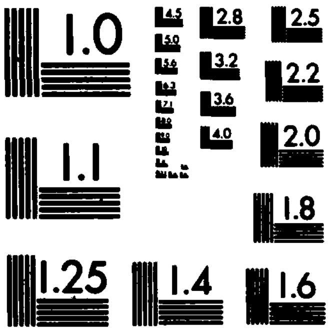
AD-A134 032 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR ADA INTEGRATED ENVIRONMENT..(U) INTERMETRICS INC CAMBRIDGE MA 05 NOV 82 IR-677-2 F30602-80-C-0291

2/2

UNCLASSIFIED F/G 9/2 NL

END

FILMED



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Several optimizations of this structure are planned. These include sorting the map ranges so that only the upper bound is required and moving the exception handler code before the declaration code to consolidate these two ranges.

An example of several VCODE and CODEGEN templates for logical OR follows. The structures are somewhat simplified to aid in clarity. These templates are diagrammed in three ways, first symbolically which is probably the clearest for intuitive understanding, second in terms of IMD, the intermediate machine description used to manually create the templates, and finally in the form of the internal representation used by the compiler and produced by the table builder program from the IMD description.

```
Symbolic representation:

OR Tree:
          :=          OR  $1,$2

         /
$1:RegF   bl_or          -- RegF stands for full word register
                         -- bl_or is the BILL "or" operator
            /
   $1:RegF   $2:RegF
--------------------------------------------------------------
IMD representation:

    production
     type: value
     pattern: $0:RegF := "bl_or" $1:ReglF $2:RegF /same01
     action: emit OR $1,$2;
    end production
--------------------------------------------------------------
Internal representation

VCODE_Rule
 -------------
| kind:value  |
| pattern     |--->(p)
| Action      |---------> TN
| Rule_id: id |          ---------------
 -------------          |Storage: RegF  |
                        |Match: id      |
                        |   ...         |
CODEGEN_Rule             ---------------
 -------------
| kind:value  |
| pattern     |--->(p)
| Action      |--------->Code_node
| Time: xx    |          ---------------
| Space: yy   |         | Op: OR        |
 -------------          | n_operands:2  |
                        | Operand   $1  |
                        | Operand   $2  |
                        | ...           |
                         ---------------
(p) --->  ----------           ----------           ----------
         | Op: bl_or|   |--->| Op: -     |   |--->| Op: -     |
         | Id:1     |   |    | Id:2      |   |    | Id:3      |
         | Parent:1 |   |    | Parent:1  |   |    | Parent:1  |
         | Next:    |---     | Next:     |----    | Next: -   |
         | AM:RegF  |        | AM:RegF   |        | AM:RegF   |
         | Leaf:0   |        | Leaf:1    |        | Leaf:2    |
         | ...      |        | ...       |        |  ...      |
          ----------           ----------           ----------
```

There will be several patterns for OR based on the different access modes for the second operands. Some of these are shown below. (There will also be flow value templates for OR not shown).

```
O Tree  :=        O  $1,$2

          /
$1:RegF   bl_or

            /
  $1:RegF   $2:MEMF


OI Tree  :=       OI  $1,$2

          /
$1:RegF   bl_or

            /
  $1:RegF   $2:Literal range 0..127
-----------------------------
OC Tree :=        OI  $1,$2

          /
$1:MemF   bl_or

            /
   $1:MemF   $2: MemF/ size < 128 bytes, same size as $1, no overlap
with $1
```

These templates are sorted by the table builder program to place the special case OI first since it matches a larger subtree. A failure link is generated from this literal directly to the second operand of the OR tree and then O tree. This is because if the tree matched this far in the OI subtree, it will also match here (since they are identical). For VCODE, the OR and O tree patterns will be merged since they differ only in their register usage. The OC tree will not be merged since it has additional restrictions.

The following diagram summarizes this information:

```
----------------------------------------------------
-- Static data area follows
----------------------------------------------------
STATICD DSECT
*
FRMSIZE DS      F        size of stack frame
HRDSIZE DS      F        size of frame header
EXCMAP  DS      A='AMAP' address of exception handler map
----------------------------------------------------
-- Code (instructions and literals) follows
----------------------------------------------------
BODY    EQU     *        entry point to the unit
        1st instruction of unit body
        etc.
        ...
        unit epilog  (wait for tasks, restore registers, return)
        ...
HANDLE  EQU     *
        1st instruction in handler (if any)
        ...
        handler epilog
----------------------------------------------------
-- Handler map follows
----------------------------------------------------
AMAP    EQU     *
        DS      A        1st instruction of declarations
        DS      A        last instruction of declarations
        DS      A=RTSRAISE  runtime raise routine
        DS      A        1st instruction of statements
        DS      A        last instruction of statements
        DS      A=HANDLE Exception handler for this range.
        DS      A        1st instruction in range of exception
                          handlers
        DS      A        last instruction in exception handlers
        DS      A=RTSRAISE
        ...              additional handler map information if
                          hoisted
                         exception handlers (due to blocks, inlines,
                           etc.)
```

3.3.3.4.4 Special Requirements

CODEGEN will process 24,000 statements/minute. The design decisions against having pseudo target code for the templates paged VMM objects were made to improve speed. Matches made during VCODE that are marked in the BILL tree can also speed pattern matching. The use of failure links avoids re-matching similar sub-patterns.

3.3.3.5 FINAL

The purpose of the FINAL phase is to perform peephole optimization and branch resolution, and to generate the object module, and address information required by DBUG and the run-time system.

3.3.3.5.1 Inputs

The inputs to FINAL are the pseudo-target code output by CODEGEN and the optimization, DBUG, and listing options that were input to the compiler DRIVER and that are encoded as BILL/BILLET attributes. All nodes of the pseudo-target code are processed.

3.3.3.5.2 Processing

There are four subphases to the processing in FINAL; one (peephole/planned) which requires a scan of all jump instructions and three (peephole/table-driven, branch resolution, and object generation) which require a pass over the pseudo-code output by CODEGEN.

(a) Peephole. This module performs a collection of peephole optimizations. That is, it looks for sequences of instructions that are close (either statically or dynamically) and that can be replaced by better sequences. There are two kinds of optimizations attempted unless OPTIMIZE(NONE) is specified.

(1) Planned Algorithms

These optimizations deal with jumps. Jumps to a location containing a jump are replaced with a single jump. Cross jumping to eliminate duplicate code sequences at the tails of then and else clauses is performed if OPTIMIZE(SPACE) is specified. Unreachable code after an unconditional jump is removed. Processing is performed by scanning the list of all jump instructions generated by CODEGEN.

(2) Table-Driven Matches

These optimizations are performed by matching templates with the code stream and replacing matched sequences with improved code. Such matches include: elimination of unreachable code appearing after an unconditional jump; eliminating jumps to the next location; replacing a sequence of shifts with one shift; and a variety of target-specific optimizations. By its very nature, this module is highly target specific in its details. Many of the optimizations have counterparts on most machines; some are discovered only by looking at the generated code and finding weaknesses. The essential ingredient here is that the optimizations are table driven and hence easily extendable. Processing is performed by a single scan of the entire list of pseudo-code produced by CODEGEN is reverse execution order.

(b) <u>Branch resolution</u>. Branch resolution is different for the IBM 370 and the PE 8/32. The PE 8/32 has three sizes of branch instruction. At code generation time, the length of the branch is unknown. This module determines the use of long and short branches using a simple algorithm. Let min and max be the minimum and maximum branch distance. Then a branch has determinate length if one of the following conditions holds:

max $\leq$ 30	(SF format)
30 $<$ min and max $<$ 32768	(RX2)
32768 $\leq$ min	(RX3)

The length of the branch instruction is 2, 4, or 6 bytes, respectively. All indeterminate length branch instructions are placed on a work list. Each branch on the work list has its maximum and minimum branch length computed. If the branch is determinate, it is removed from the work list. Determining the length of one branch instruction may cause other branch instructions to become determinate in length. The work list is repeatedly scanned until a pass causes no branches to be removed. All remaining branches are assigned a length corresponding to the minimum. This algorithm converges rapidly and seldom requires more than a few iterations. Furthermore, most of the branches are resolved on the first iteration, leaving little work for subsequent iterations. Lastly, the work involved after the first pass is proportional to the number of remaining unresolved branches.

The IBM 370 has no self-relative branches. All branch locations must be reached by base register plus displacement. It is both undesirable and unnecessary to reserve a large number of base registers to address the program. It is undesirable because the same general registers are more profitably used for data items. It is unnecessary because most branches do not span large distances.

The program is divided into sections, each of which can be addressed with a single program base register. Branches within a program section are implemented by direct branches. The initial instruction of each section loads the program base register with the address of the section, as do all inter-section branches. Processing is performed by a single execution order scan of the pseudo-code produced by CODEGEN.

This method is similar to that used by Scarborough and Kolsky, differing only in using one register instead of two or three. Their experience indicates that the gain in register use for data items exceeds the cost of section address loading.

On the IBM 370 there is no general purpose immediate addressing mode. (The load address (LA) instruction is a useful exception.) FINAL allocates short literals (e.g., address constants) in pools for each segment. Therefore, each segment contains both its own code and the pool of literals used by that segment. The pool is placed at the end of the segment.

On the PE 8/32, which has immediate instructions and 24 bit displacements, literal values whose only references are via immediate instructions are removed from the literal pool.

(c) Object generation. The doubly-linked list of pseudo-code is traversed and the following actions performed:

(1) each instruction is converted into object module format;

(2) if an assembly listing has been requested, an assembly listing line is generated.

(3) if DBUG(BREAK) is specified, each statement node is converted into an entry in the hook table; and the table of statement addresses is built. This hook table is either included in the object module or attached to the DIANA corresponding to the BILL unit depending on the requirement of DBUG.

3.3.3.5.3 Outputs

The outputs from FINAL are the object module, including the handler maps and list sequences required by the run-time system (see AIE(1).KAPSE(1)), and if DBUG (BREAK), the hook table required by DBUG. Additional information will be generated, if required by the linker, to associate external symbols with DIANA.

3.3.3.5.4 Special Requirements

FINAL will process 12,000 statements/minute. The list of all jump instructions produced by CODEGEN was designed to speed this processing.

3.3.3.6 UTILITIES

The UTILITIES package contains comman routines required by the Back End. These will be defined when the lower level structure of the phases is known.

3.4 Adaptation

The compiler includes specific strategies which are appropriate for the PE 8/32 and IBM 370 architectures.

The IBM 370 is very similar to the PE 8/32 in many respects. However, the IBM 370 addressing modes cause special problems. The limitation of 12 bit displacements to byte addresses, the limitations on immediate operands, and the lack of program counter relative addressing mean that all references to objects (code, variable, literals, etc.) require that a register be loaded with a base address. This design attempts to minimize the need for different base registers and reduce the cost of base register loading.

During FLOW a choice is made between strength reduction and base register optimization. The base register optimization combines the constant components of addressing computations and uses a single base register for each set of references which is in a different block of 4096 bytes. FLOW rearranges the execution order to reduce the number of registers which are simultaneously needed. VCODE chooses code which reduces the demand for registers. For example, choosing a shift, instead of a multiply by a power of two, reduces register requirements from an even/odd pair to a single register. In TNBIND base addresses compete with values for register space. TNBIND chooses a balance between the use of registers for base addresses and for values. A code region which has more references to value TNs and fewer to base register TNs will have more registers allocated for values, and conversely. This is to be contrasted with a design which allocates a fixed set of registers for bases. TNBIND is responsible for allocating the even and odd registers and register pairs which are required; e.g., for integer multiply and divide (for both the IBM 370 and PE 8/32). Some function results are returned in registers. TNBIND attempts, where possible, to use the value returned without first moving it to another register or to storage.

For the IBM 370, FINAL divides the code into segments, each of which can be accessed with a single base register. Each segment includes a pool of short literals (including address constants) used within the segment. Thus, within-segment branches and short literal references are made relative to a single base register. A separate control section is generated for each subprogram; thus, subprograms that are not called need not be linked in. For the PE 8/32, FINAL chooses immediate mode and discards literals which are only so referenced. FINAL also optimizes branches to use the short relative branches available on that machine.

3.5 Capacity

The contract specifies a number of performance requirements. For the compiler these are:

(1) The compiler is required to compile a single 1000 statement Ada program within one cpu minute on the IBM 4341 installed at Intermetrics with four users logged on, one running the compiler and the others running the command processor or debugger or editor. The compiler time is measured with the options for LISTING, OPTIMIZATION, COMMENTS, and DBUG turned off.

(2) The compiler shall require no more than 512K bytes of main memory for any Ada program. Up to an additional 512K bytes, if available, may be utilized to speed up compilation of larger programs.

To achieve the speed requirements, we shall require that each section of the compiler use one third of the total time available. Therefore:

(1) The Front End must process 3000 statements/cpu-minute into DIANA.

(2) The Middle Part must process the resulting DIANA in one cpu-minute.

(3) The Back End must process the resulting BILL in one cpu-minute.

(4) Listing generation is a separate timing issue.

(5) Linking shall take no more than 20% of the time taken by the Back End. This means that the compiler compiles at the required speed with optimization, and that the compiler and linker, together, without optimization, perform as required.

To achieve compiler size requirements, we shall require that DRIVER, resident run-time system, VMM support routines, and the largest compiler phase fit within 300K bytes, leaving 212K bytes for paged and non-paged data.

The run-time system (without tasking) is constrained to 15K bytes and VMM is constrained to 100K bytes.

The semantics phase of the Front End is estimated to be the largest phase, and will be restricted to 180K bytes. All other phases are constrained to be less than 180K bytes.

The DRIVER is to be less than 5K bytes.

A number of implementation decisions will be based upon the performance requirements placed upon the compiler as a whole. In order to evaluate, in advance, the costs of various strategies that the Front End might use, a performance model has been built. This model covers factors about the Ada language, the target machine, and the actual implementation.

For example, factors involved in parsing include:

(1) size of input buffer
(2) avg. characters per token
(3) avg. tokens per reduction
(4) avg. DIANA nodes per reduction
(5) % of declarations per compilation unit
(6) avg. size of DIANA declaration node
(7) size of VMM paging space
(8) I/O overhead per line read
(9) avg. characters per line
(10) instruction overhead per Ada procedure call

These are combined into a linear equation which estimates the processing taken under a variety of assumptions. Similarly, the semantics model incorporates symbol table hit ratios, tree walk overhead, etc. Values of the parameters are either "best-guess" estimates, or measurements taken from other compilers, including the Ada bootstrap compiler.

The results of performance modeling guide current design. In addition, during implementation, extensive measurements of early coding will be taken to increase the accuracy of our picture. These actual measurements will be used to revise the design before complete implementation has occurred.

The requirements of generated code efficiency, retargetability, rehostability, statistics gathering, and compile time efficiency interact and involve trade-offs from time to time. To clarify our underlying philosophy for various trade-offs, our priorities for the compiler are as follows:

(1) Maintainability and retargetability/rehostability

(2) Code efficiency

(3) User friendliness (including compile-time speed)

The Front End uses some internal stacks to aid parsing. The needed stack space varies with the language construct being parsed and the nesting depth of the construct within the compilation unit. A FATAL error message shall be generated if the size of a parse stack is exceeded.

There are two such stacks, the PARSER stack and the STATE stack. They are each limited to 200 elements.

The Front End inputs Ada source lines via the KAPSE into a local I/O buffer. A source line exceeding 255 characters will cause a SEVERE error message to be generated.

To improve performance, various compiler phases attempt to keep some kinds of data only in core, and not paged to a file. This restricts the size of user programs. The exact restrictions will be specified at a later time but; a rough estimate is that the user is limited to 2000 lines of Ada source PER COMPILATION UNIT.

Additional size restrictions are:

(1) In the Back End, the total size of BILL nodes to represent a subprogram body may not exceed a figure to be determined.

(2) In the Front End, the size of the DIANA for a compilation unit may not exceed a figure to be determined.

(3) VMM limits the compiler to 200 subdomains accessible at once. This limits the number of units that may be WITHed, including chains reaching units the WITHed unit WITHed (implied references). This also limits the nesting depth of SUBUNITS. The total nesting depth + implied and explicit WITHs + 1 must be less than 200.

(4) VMM limits the total number of subdomains to 32K. This limits the number of objects in the program library to 64K as well. An object is: an abstract syntax object, a DIANA object, a listing object or the machine code object. Recompilation generates new DIANA, listing and machine code objects. Objects may be thrown away and reclaimed, making available more objects.

(5) VMM limits the total number of bytes of a subdomain. Therefore, no compilation unit's DIANA may exceed the VMM restriction, currently 2**29K bytes.

4.0 QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Compiler testing will be conducted in four stages. Stage one is subprogram testing, which tests each CPC and its subunits. Stage two is program testing, which tests the Front End (COMP.FE), the Middle Part (COMP.MID), and the Back End (COMP.BE) individually. Stage three is subsystem testing, which tests the entire compiler. These tests verify that the compiler as a whole accepts Ada source programs and produces the corresponding object code for that program, each phase cooperating with the other phases. Stage four is integration and acceptance testing, which validates the compiler performance for purposes of delivery. Stage four includes both integration and acceptance tests because the compiler must be verified in combination with other MAPSE tools (debugger, linker, and recompilation checker) that depend upon its output.

4.2 Test Requirements

This section describes the requirements for subprogram level, program level, and subsystem level testing of the compiler. The discussion includes tools, facilities, and techniques, both automatic and manual, for performing the tests.

4.2.1 Subprogram Testing

Subprogram testing consists of testing all subunits of the CPCs as well as testing the individual CPCs. These tests wll be designed to verify the C-5 specification for each CPC. The tests will be designed and executed by implementation personnel. Test descriptions and test results for each CPC will be submitted to Quality Assurance (QA).

Testing at the subprogram level is partially automated (the compile-time checking performed on the subprogram) but most of this testing is manual. Prior to this testing, both the code author and a designated code reader for the phase will inspect the subprogram, with particular attention to its correctness, readability, and efficiency.

4.2.2 Program Testing

The compiler Front End (COMP.FE), Middle Part (COMP.MID) and Back End (COMP.BE) will each be tested individually. The tests will be described in detail in test descriptions which will be submitted to QA and formal test reports will be issued.

For COMP.FE, these tests verify that COMP.FE accepts the full Ada language and, for each source construct, produces the corresponding DIANA. Testing will be conducted by first compiling simple programs and then successively testing with more complex programs as the implementation becomes more complete. Thus, the emphasis is on processing complete programs as early as possible, successively adding compiler portions to process more complex programs. The DIANA for these tests will be inspected manually for correspondence with the source language.

Tests include both legal and illegal Ada programs. Legal tests check that no error messages are generated and that appropriate DIANA is produced. Illegal tests check that all errors contained in the source program are detected by the compiler.

For COMP.MID, these tests verify that COMP.MID accepts DIANA and, for each DIANA construct, produces the corresponding BILL. For COMP.BE, these tests verify that COMP.BE accepts BILL and, for each BILL construct, produces the corresponding IBM 370 (or PE 8/32) object code.

Because the compiler phases communicate through intermediate languages, phases after COMP.FE may be tested independently. COMP.FE is tested using Ada source programs as input and inspecting the resulting DIANA for correspondence with the source program. Testing of COMP.MID and COMP.BE need not wait until COMP.FE is fully operational. Intermediate language programs may be edited manually, using the Virtual Record Notation input/output packages in VMM.VMM, and such programs allow testing to proceed in parallel. For example, EXPAND is tested independently of STORAGE by editing STORAGE attributes into DIANA trees manually in human-readable form and using the VMM VRN input package to convert the DIANA into internal form for processing by EXPAND.

4.2.3 Subsystem Testing

Subprogram tests for the compiler will be described in formal Test Procedures. Subsystem tests include the applicable suite of Ada Compiler Validation Capability (ACVC) tests. Ths suite includes all ACVC tests except those that are concerned with inapplicable machine dependent tests. Prior to formal validation, other tests for the full compiler will be constructed to verify correct processing for significant Ada constructs. The formal subsystem tests for the IBM 370 will be conducted after the compiler is complete and has been compiled by the bootstrap; they will be conducted again after the self-host. For the PE 8/32, the tests will be conducted when the PE 8/32 is complete and the KAPSE has been rehosted.

4.3 Acceptance Test Requirements

Acceptance tests will be conducted to ensure that the compiler conforms to its general requirements. The ACVC tests as well as the Ada compiler itself and all other AIE MAPSE tools will be used as acceptance tests. These tests concern the speed of compilation, diagnosis and useful classification and reporting of errors, and simplicity and usability of the user interface. Speed tests will be measured formally by determining the speed (in lines per minute) of the compiler. The evaluation of error handling and the usability of the user interface is measured informally.

4.4 Facilities

4.4.1 Bootstrapping Requirements

To test the IBM 370 compiler, which is an Ada program, an existing Ada compiler, linker, and DBUG must be available. This existing Ada compiler, called the "bootstrap" compiler, executes on the IBM 370, accepts a sequential subset of Ada, and generates code for the IBM 370. The AIE compiler is written and tested using this Ada subset.

Because the bootstrap is an interim tool, the bootstrap omits many code optimizations included in the AIE compiler. Therefore, to enhance performance, the AIE compiler must be used to compile itself. Once the AIE compiler is sufficiently operational, the AIE compiler source code is translated by the executable AIE compiler, which was translated using the bootstrap. The AIE compiler for the PE 8/32 is developed using the AIE compiler running on the IBM 370, without use of the boostrap.

4.4.2 Metering

To verify that the compiler satisfies the speed requirements (1000 statements per minute), a timing package is available to be included within the compiler. For the purpose of measuring compiler performance, a "statement" is considered to be a declaration, a statement, or a representation clause node in the DIANA tree. These statement counts are collected following the GENINST phase. Because generic instantiation can result in multiple bodies, the representation after GENINST gives a more accurate measure of performance than the representation after semantic analysis. The CPU time for each phase is determined and is output in the statistics section of the listing.

4.4.3 Test Scripts

During the period that the MAPSE command processor is unavailable, the compiler wll be tested using the command language facilities on the IBM 370. When it is available, testing will make use of MAPSE command language scripts where appropriate. To test the entire compiler in conjunction with the linker and the run-time system, test scripts to perform "compile, load, and go" functions will be used. Other scripts to perform regression testing and component testing also will be constructed as needed.

APPENDIX A: ERROR MESSAGES

A.1 General Format

The general format error messages is:

<stmt> <severity> <code> <phase> <message>

where: <stmt> is the statement number, if applicable
<severity> is severity level
<code> is the nubmer of the diagnostic message
<phase> in which the error occurred
<message> is the English text of message

The places where error message are generated for user errors are:

LEXSYN: for bad syntax
PRE-SEMANTICS: for bad semantics
SEMANTICS: for bad semantics
STORAGE: for bad pragmas and representation specifications

A.2 Severity Levels

FATAL: The compiler is aborted due to drastic errors.

INTERNAL: An internal compiler error occurred which may or may not cause a function of the compiler to terminate abnormally. If the compiler is able to continue, normal outputs may or may not be produced.

ERROR: The user's program has an error. Processing continues, but normal outputs may or may not be produced.

WARNING: The compiler has discovered a situation the user may be unaware of, but which is legal. Normal outputs are produced.

NOTE: Advisory information is given to the user, often for the purpose of suggesting optimizations to his program.

A.3 Error Messages Generated by LEXSYN

errors in basic character syntax (severity = ERROR)

```
INVALID PRINTING CHARACTER xx
INVALID NON-PRINTING CHARACTERS HEX "xx"
INCORRECT USE OF UNDERSCORE
LINE LENGTH OVERFLOW
QUOTE (") INVALID IN STRING
CHARACTER LITERALS MISSING END QUOTE ON SAME LINE
TAB IN STRING
```

PERIOD (.) FOUND IN SPECIALS FOLLOWED BY ZERO
EXPECT DOUBLE (") NOT SINGLE QUOTE (') TO DELIMIT STRING

errors in numeric syntax: (severity = ERROR)

BASE VALUE (2 OR > 16 NOT ALLOWED
DIGIT xx OUT OF RANGE FOR BASE yy
NO DIGITS FOLLOWING BASE
BASED NUMBER NOT TERMINATED IWTH # OR :
EXPONENT WITHOUT MANTISSA RADIXPOINT
EXPONENT WITHOUT DIGITS
NUMBER SHOULD BE SEPARATE FROM ADJACENT FOLLOWING IDENTIFIER
DIGIT NEEDED AFTER AND BEFORE RADIXPOINT
MULTIPLE RADIXPOINTS
NO BASE ON BASED NUMBER

general token errors: (severity = ERROR)

xx EXPECTED BEFORE THIS TOKEN
xx EXPECTED AFTER THIS TOKEN
xx EXPECTED INSTEAD OF yy
xx EXPECTED INSTEAD OF yy zz
UNEXPECTED xx
RESERVED WORD xx MISSPELLED

secondary error recovery: (severity = ERROR)

BAD COMPILATION
BAD DECLARATION
BAD STATEMENT
BAD EXCEPTION HANDLER
BAD EXCEPTION CHOICE
BAD PARAMETER DECLARATION
BAD GENERIC FORMAL
BAD COMPONENT DECLARATION
BAD GENERIC FORMAL
BAD COMPONENT DECLARATION
BAD ENTRY DECLARATION
BAD CASE ALTERNATIVE
BAD CHOICE
BAD EXPRESSION
BAD TERM
BAD FACTOR
BAD CONDITION IN IF STATEMENT
BAD SELECT ALTERNATIVE LIST

corrective actions taken: (severity = ERROR)

xx INSERTED
xx INSERTED TO MATCH yy
xx DELETED
xx SPELLING CORRECTED

capacity limit reached:

```
PARSE STACK OVERFLOW (severity = FATAL)
LINE BUFFER OVERFLOW (severity = INTERNAL)
```

A.4 Error Messages Generated by Presemantics and Semantics

Error messages generated by semantics have not been formalized yet. The intent is to match the messages provided by the ALS, in order to gain some compatibility.

A.5 Error Messages Generated by Storage

Error messages generated for bad pragmas and representation specifications have not been formalized yet.